Implementation Variants for Position Lists

Andreas Schmidt*†, Daniel Kimmig†, and Steffen Scholz†
* *Department of Computer Science and Business Information Systems,*
Karlsruhe University of Applied Sciences
Karlsruhe, Germany
Email: andreas.schmidt@hs-karlsruhe.de
† *Institute for Applied Computer Science*
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: {andreas.schmidt, daniel.kimmig, steffen.scholz}@kit.edu

Abstract*—Within “traditional” database systems (*row store*), the values of a tuple are usually stored in a physically connected manner. In a *column store* by contrast, all values of each single column are stored one after another. This orthogonal storage organization has the advantage that only data from columns which are of relevance to a query have to be loaded during query processing. Due to the storage organization of a *row store*, all columns of a tuple are loaded, despite the fact that only a small portion of them are of interest to processing. The orthogonal organization has some serious implications on query processing: While in a traditional *row store*, complex predicates can be evaluated at once, this is not possible in a column store. To evaluate complex conditions on multiple columns, an additional data structure is required, the so-called *Position List*. At first glance these *Position Lists* can easily be implemented as a dynamic array. But there are a number of situations where this is not the first choice in terms of memory consumption and time behavior. This paper will discuss some implementation alternatives based on (compressed) bitvectors. A number of tests will be reported and the runtime behavior and memory consumption of the different implementations will be presented. We additionally extended the existing WAH library for compressed bitvectors by a number of new methods to be used for the purpose of implementing the functionality of *Position Lists* based on (compressed) bitvectors. Finally, some recommendation will be made as to the situations in which the different implementation variants for *Position Lists* will be suited best. Their suitability depends strongly on the selectivity of a query or predicate.

***Keywords*–Column stores; PositionList implementation variants; bitvector; compression; run length encoding; performance measure**

I. Introduction

This article is an extended version of a paper entitled *Considerations about Implementation Variants for Position Lists* [1] presented at the Fourth International Conference on Advances in Databases, Knowledge, and Data Applications in Sevilla, Spain. Some important extensions include the extension of the WAH library by a number of transformation functions between different representation forms of a *Position List* as well as the implementation of a special append-based bitset function on compressed bitvectors, which allows the usage of compressed bitvectors in the typical *Position List* generation process when evaluating a single predicate.

Nowadays, modern processors utilize one or more cache hierarchies to accelerate access to main memory. A cache is a small and fast memory which resides between the main memory and the CPU. In case the CPU requests data from the main memory, it is first checked, whether these data are already contained in the cache. In this case, the item is sent directly from the cache to the CPU, without accessing the much slower main memory. If the item is not yet in the cache, it is first copied from the main memory to the cache and then sent to the CPU. However, not only the requested data item, but a whole cache line, which is between 8 and 128 bytes long, is copied into the cache. This prefetching of data has the advantage of requests to subsequent items being much faster, because they already reside within the cache. Depending on the concrete architecture of the CPU, the speed gain when accessing a data set in the first-level cache is up to two orders of magnitude compared to regular main memory access [2]. This means that when a requested data item is already in the first-level cache, the access time is much faster compared to the situation, when the data item must be loaded from the main memory (this situation is called a cache miss). The use of special data structures which increase cache locality (the preferred access of data items already residing in the cache) is called *cache-conscious* programming.

Column stores take advantage of this prefetching behavior, because values of individual columns are physically connected. Therefore, they often already reside in the cache when requested, as the execution of complex queries is processed column by column rather than tuple by tuple. This difference between a “traditional” *row store* and a *column store* is illustrated in Figure 1. In the upper part of the figure, a relation, consisting of six tuples, each with five columns, is shown. The lower part of the figure shows the physical layout of this relation on disk or in the main memory. On the left side, the row store layout is represented. The row store stores all values of one tuple in a physically connected

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

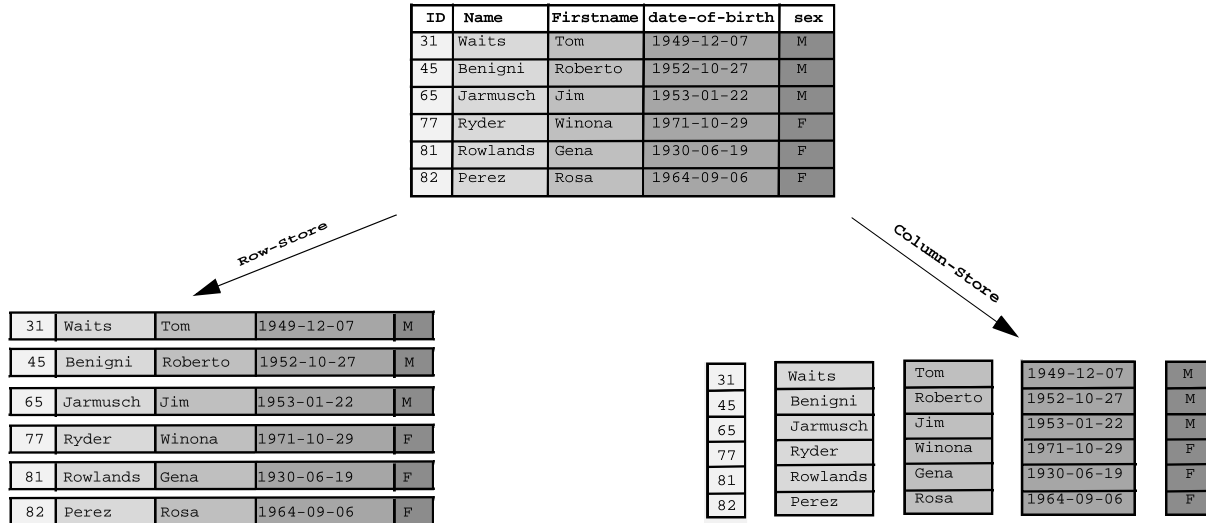

Figure 1. Comparison of the layouts of a row store and a column store (from [3])

manner. In contrast to this, a column store contains all values of each single column one after another.

This also means that the decision whether a tuple fulfills a complex condition on more than one column is generally delayed until the last column is processed. Consequently, additional data structures are required to administrate the status of a tuple in a query. These data structures are referred to as *Position Lists*. A *Position List* stores information about matching tuples. The information is stored in the form of a Tuple IDentifier (TID). The TID is nothing more than the position of a value in a column. Execution of a complex query generates a *Position List* with entries of the qualified tuples for every simple predicate.

Complex predicates on multiple columns can be evaluated in two different ways. First, as shown in Figure 2, the predicates can be evaluated separately, and in a subsequent step, the resulting *Position Lists* can be merged. The advantage of this variant is, that the evaluation of the predicates can be done in parallel. A drawback of this solution is, that all column values must be evaluated.

In contrast to this, the evaluation of the query can also be done sequentially, as shown in Figure 3. In this case, a *Position List* representing the result of a previously evaluated predicate is an additional input parameter for the evaluation of the second predicate. Not all column values have to be evaluated, but only those for which an entry in the first *Position List* exists. The drawback of this solution is the strict sequential program flow and a slightly more complex execution, which may probably cause more cache misses compared to the parallel version. Which of the variants is better suited depends on the boundary conditions of the query.

In previous work, we developed the Column Store Toolkit (CSTK) [3] and used it as a starting point for further research

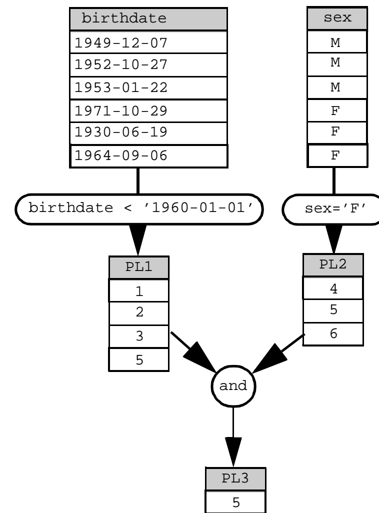

Figure 2. Isolated evaluation of predicates on their corresponding *Position Lists* and subsequent merging of the resulting *Position Lists* (from [3])

in the field of optimizing SQL queries based on a column store architecture [4].

The main objective of this paper is to present an in-depth analysis of different implementation variants of *Position Lists* and to demonstrate their advantages and disadvantages in different situations in terms of runtime behavior and memory consumption.

The paper is structured as follows. After giving an overview over related work in the next section, we will discuss some specific details of *Position Lists*. Then, the most important components of the CSTK will be introduced. After that, a number of experiments with respect to runtime behavior and memory consumption will be performed in

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

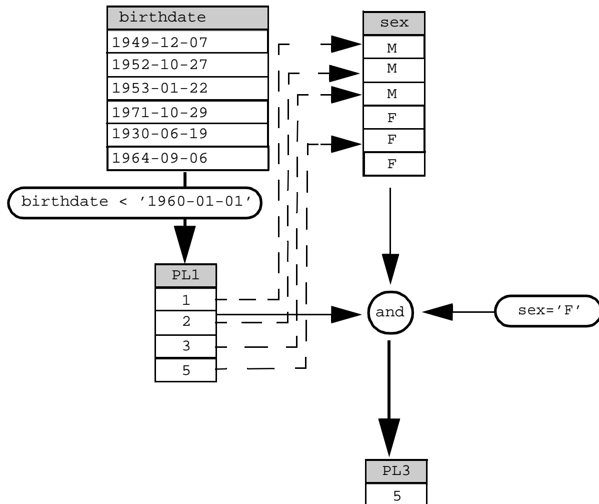

Figure 3. Iterative evaluation of predicates, using *Position Lists* as additional parameters

the main part. Finally, results will be summarized, and an outlook will be given on future research activities.

II. Related Work

First work addressing column stores (vertical storage orientation) is dated back into the 80th [5], [6]. During the last decade, a number of new research prototypes, based on a vertical partitioning of data, appeared and did show some advantages. From these systems, C-Store [7] and MonetDB [8] are the most notably ones. On the commercial side, Infobright [9], SAP-Hana [10], Sybase IQ [11], and Vertica [12] (a commercial version of C-Store), amongst others, appeared. Today, also big players like Oracle and Microsoft implemented columns store technologies into their database systems [13], [14]. Various publications compare the performance of column stores with that of row stores for different workloads [7], [15], [16]. In contrary, [17] examines different execution plan variants for column stores, while [18] considers the impact of compression. Following the work in [17], we examine different implementation variants for the underlying data structures and algorithms of the operations used in the execution plan of a query.

Abadi et. al. in [19] mention different implementation variants for *Position Lists*, i.e., a simple array, a bitstring or a set of ranges of positions, but did not compare these different solutions with respect to runtime behaviour. In contrast to the previous mentioned work, we do not use a fixed structure for implementing the *Position Lists*, but compare the runtime and memory consumption behaviour of different implementation variants with respect to the selectivity of a predicate.

III. Position Lists

From a logical point of view, a *Position List* is nothing more than an array or list with elements of the data type *unsigned integer* (UINT) as far as structure is concerned. However, it has a special semantics. The *Position List* stores TIDs. A *Position List* is the result of a query via predicate(s) on a *Column*, where the actual values are of no interest, whereas the information about the qualified data sets is desired. *Position List*s store the TIDs in ascending order without duplicates. In other words, a *Position List* stores the information for each tuple no matter whether it belongs to a result (so far) or not.

A. *Operations on* Position Lists

The fundamental logical operations on *Position Lists* are appending TIDs at the end (write operation), iterating over the list of TIDs (read operation), and performing *and/or* operations on complete lists.

Further operations that are mainly based on this basic functionality, include the materialization [17] of the corresponding values from the requested columns, the storage of the whole list or parts of it in a file, and the import from a file.

B. *Implementation Variants*

Based on the logical structure and behavior discussed above, the first intuitive implementation of a *Position List* is using a dynamic array (an array of flexible size) of unsigned integer values. The advantage of this variant is, that the implementation is straight forward and the storage of the TIDs is cache-conscious [20], [21] in the context of the above-mentioned operations like iterating, storing, and *and/or* operations.

As *Position List*s store the TIDs in ascending order without duplicates, typical *and/or* operations are very fast, as the cost for both operations is $O(|Pl_1| + |Pl_2|)$.

One big drawback of the implementation as a dynamic array is the fact, that the lists may be very large. This is especially true for predicates over multiple columns, where no predicate has a *high selectivity*. In this context, *high selectivity* means, that only a small number of tuples qualify the condition. A *low selectivity*, by contrast, means that a lot of tuples satisfy the condition. Typical predicates of low selectivity are the “family status” or the “gender” of a person. Let us consider a conjunctive query consisting of 6 predicates on different columns. Each single predicate has a selectivity of up to 50% (i.e., gender, family status, etc.). The overall selectivity of the query is about 1.5% of the original number of tuples, but the size of the cardinality of the individual *Position Lists* is up to 50% of the original table. Starting with the predicate of the highest selectivity and iteratively examining the values of tuples from the subsequent columns which qualified previously (see Figure 3) can reduce this problem. However, if no or only vague

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

information about the selectivity of the different predicates is available, this can be a serious problem. Figure 4 shows the size of a *Position List* in megabytes with respect to the selectivity of the predicate for a 100 million tuple table. In the worst case, the resulting *Position List* can be bigger than the original column (e.g., for columns with binary values or a small number of possible values only).

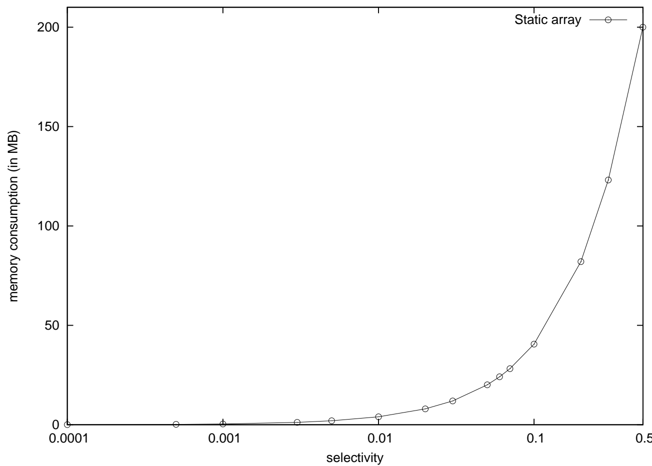

Figure 4. Memory consumption of a *Position List* implemented as array (logarithmic scale on x-axis)

The problem of the unpredictable size of the intermediate *Position Lists* can be prevented by using a bitvector to represent the *Position List*. Here, every tuple is represented by one bit. A value of '1' means that the tuple belongs to the (intermediate) result, a value of '0' means that the tuple does not belong to the result.

This has the advantage of the size of a *Position List* being exactly predictable, independently of the selectivity of the predicate. The selectivity only has impact on how many bits are set to '1'. Moreover, the two important operations *and* and *or* can be mapped on the respective primitive processor commands, which makes the operations fast. If *Position List*s are sparse, bitvectors can also be compressed very well using run length encoding (RLE) [22]. The idea behind RLE encoding is that if only a small number of bits are one, the '0' bits are not stored physically, but only the number of '0' bits are stored.

Figure 5 presents a simple example of this principle.

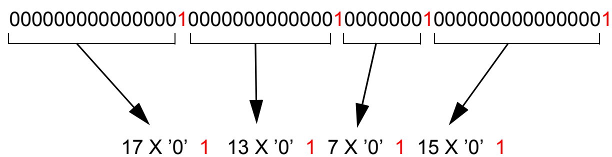

Figure 5. Principle of run length encoding (RLE)

The *Word Aligned Hybrid* algorithm (WAH) [23] uses this principle and distinguishes between two word types: *fills* and *literals*. The two word types are distinguished by the most significant bit, so 31 (63) bits remain for the stored bits per word or the length field. A *literal* is a word consisting of 31 (63) bits, of which at least one bit is '1'. A *0-fill* consists of a multiple of 31 (63) '0' bits which are stored in one word. The maximum number of '0' bits which can be stored in one word is $31 * 2^{31}$ (resp. $63 * 2^{63}$ for the 64-bit version).

The necessary operations like iterating, *and*, *or* can be performed on the compressed lists, thus avoiding a temporary decompression of the compressed representation. In the context of this paper, the bitvector implementation of the WAH algorithm and a simple plain uncompressed bitvector implementation are used. The WAH algorithm is considered to be one of the fastest algorithms for performing logical *and/or* operations on compressed bitvectors.

IV. The Column Store Toolkit

The Column Store Toolkit (CSTK) was deveolped as a toolkit with a minimum amount of basic components and operations required for building column store applications. These basic components were used as catalysts for further research into column store applications and for building data-intensive, high-performance applications with minimum expenditure.

The main focus of our components is on modeling the individual columns, which may occur both in the secondary store as well as in main memory. Their types of representation may vary. To store all values of a column, for example, it is not necessary to explicitly store the TID for each value, because it can be determined by its position (dense storage). To handle the results of a filter operation, however, the TIDs must be stored explicitly with the value (sparse storage).

Another important component is the already discussed *Position List*. Just like columns, two different representation forms are available for main and secondary storage. In this paper, it is concentrated on the main memory behavior of the *Position Lists*.

To generate results or to handle intermediate results consisting of attributes of several columns, data structures are required for storing several values (so-called multi-columns). These may also be used for the development of hybrid systems as well as for comparing the performance of row and column store systems.

The operations mainly focus on writing, reading, merging, splitting, sorting, projecting, and filtering data. Predicates and/or *Position List*s are applied as filtering arguments.

Figure 6 presents an overview of the most important operations and transformations among the components. The arrows show the operations among the different components (ColumnFile, Dense-/Sparse ColumnArray, PositionList, and PositionListFile). For a detailed description of the operations, see [3].

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

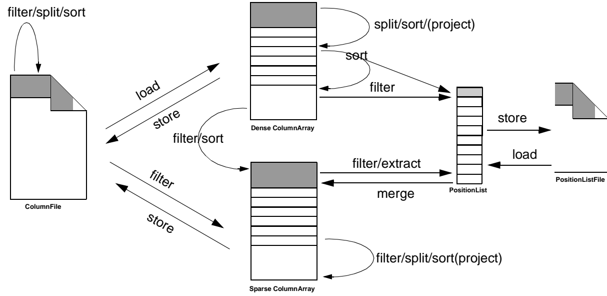

Figure 6. CSTK: Components and operations (from [3])

V. MEASUREMENTS

A. Elemental Operations

1) Experimental setup: All tests were performed on a 64-bit laptop running Windows 7 Enterprise with an Intel(R) Core(TM) i7-3520M CPU @ 2 x 2.90 GHZ and 8 GB of RAM. The used C++ compiler was gcc 4.5.3.

2) Memory consumption: In a first experiment, we compare the size of the different data structures with respect to memory consumption. As shown in Figure 7, the behavior of the array implementation is quite good for very high selectivities (0.01 and below), but changes for the worse at medium and low selectivities. Uncompressed bitvectors (plain bitvector, WAH-uncompressed) behave independently for all selectivities, their size is determined by the number of tuples in a table only. Compressed bitvectors show a very good behavior for all selectivities. If the selectivities get low, they behave like uncompressed bitvectors (compared to a pure uncompressed implementation of a bitvector, there will be a slight overhead of 1/32 resp. 1/64.). From a selectivity of about 3%, the array has a higher memory consumption than the uncompressed bitvector.

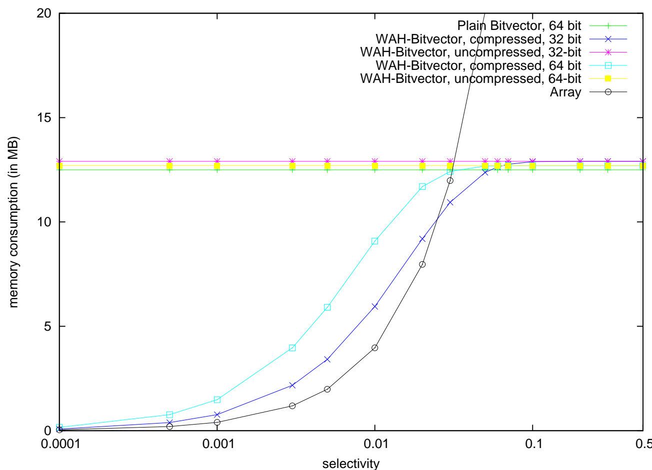

Figure 7. Memory consumption of different implementation alternatives for *Position Lists*

3) Iterating over TIDs: In the next experiment, we examine the runtime behavior of the two elemental operations:

- Appending TIDs on a *Position List*
- Iterating over the TIDs in a *Position List*.

These two operations are heavily used in the implementation of the CSTK components.

We implement a simple bitvector class on our own (without compression facility) and also use the well-known WAH algorithm. The overhead of the uncompressed representation of WAH is quite small in terms of both runtime and memory consumption.

In contrast to the original implementation of the WAH algorithm, we also use hardware support for special operations. The Leading Zero Count Instruction (LZCNT) is used to find the '1' bits inside a processor word. This leads to a performance advantage of a factor of 3 compared to the orginal WAH version.

In our first experiment, we take a table of 100 million tuples and formulate predicates with different selectivities between 0.0001 and 0.5. The TIDs of the qualified tuples are then stored in the different representation forms (plain bitvector, WAH bitvector uncompressed/compressed with 32 and 64 bit word size, array). After that, we measure the time to iterate over all the stored TIDs.

Figure 8 presents an overview of the runtime behavior for our different implementations:

The fastest implementation for all selectivities is the dynamic array. In contrast to this, the worst runtime behavior is reached by the standard WAH iterator (both 32- and 64-bit version), which therefore will not be considered any further. More interesing values come from the iterators which use the *__builtin_clzl* instruction from the gnu compiler family, which is mapped on the LZCNT instruction, if available (the plain bitvector implementation is the fastest).

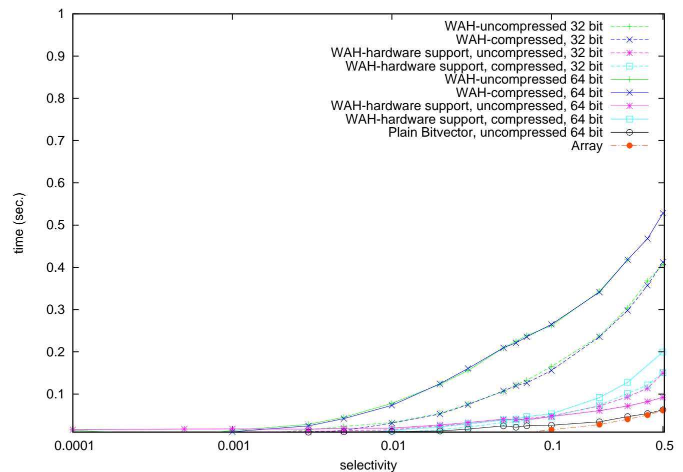

Figure 8. Measured time to iterate over 100 million data sets with different selectivities

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Two more detailed graphs are given in Figure 9 and Figure 10. Here the static array implementation and the LZCNT-supported iterators are considered for high and low selectivity, respectively.

While Figure 9 shows the details for selectivities between 0.0001 and 0.05, Figure 10 shows the lower selectivities between 0.05 and 0.5. One interesting point is, that with low selectivity (Figure 10) the hardware-supported iteration behaves differently for the 32- and 64-bit WAH version. While the compressed version is faster for the 32-bit version, the opposite is true for the 64-bit version. This behavior can be found with the better compression ratio of the 32-bit version for lower selectivities, which leads to a smaller amount of memory which has to be loaded into the CPU.

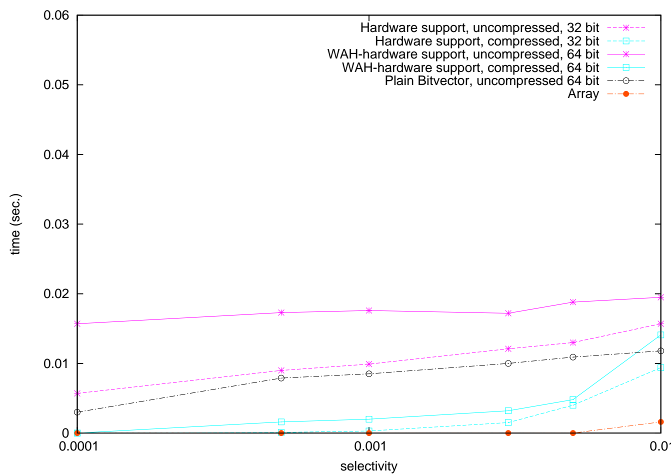

Figure 9. Measured time to iterate over 100 million data sets with high selectivity

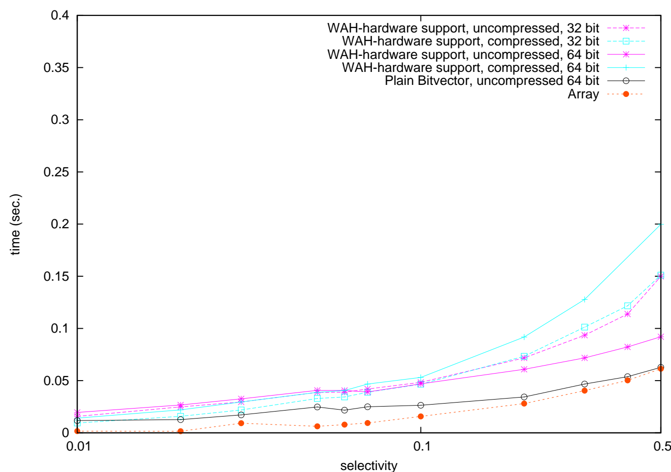

Figure 10. Measured time to iterate over 100 million data sets with low selectivity

It is obvious that the time for the uncompressed bitvector versions is the least dependent on the selectivity. This can be explained by the dominating time for loading the data from the main memory into the CPU. For all other implementations the influence of the descending selectivity is higher.

Although the static array implementation is faster by a factor of five for some selectivities, we also have to consider that in absolute values, the time of iterating over a bitlist of 50 million entries (selectivity: 0.5) is between 0.08 seconds (array) and 0.26 (64-bit, hardware-supported, uncompressed). This is not bad and probably not such a dominating factor compared to the memory consumption of the different implementations shown in Figure 7.

4) Writing TIDs: In our next experiment, we analyze the time to write TIDs in the different implementation variants. This operation is done every time, when a predicate is evaluated against a column value and found to be "true".

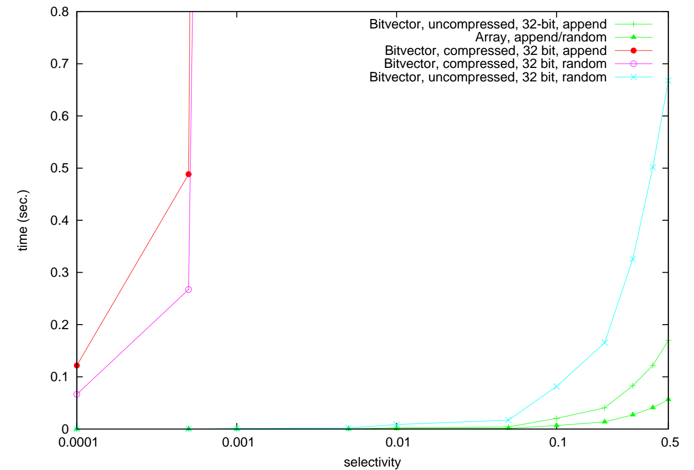

Figure 11. Measured time to write TIDs in different implementation variants for *Position Lists*

As a basic condition we can assume that writing of TIDs is mostly done in the append mode. The reason is that when evaluating a predicate on a column, this is done sequentially value by value with increasing TID values. In some complex situations, however, TIDs must be written in random order (i.e., after a previous sort operation on a column).

The results for this experiment are shown in Figure 11.

We assume 100 million datasets and measure the time to set a number of TIDs for different selectivities. So for example for a selectivity of 0.5, we have to write 50 million TIDs.

Again, the storage as an array of UINT values is the fastest solution for all selectivities. This is true for the append mode and the random order mode (from the implementation point, there is no difference between the two variants). The uncompressed bitvector turned out to be the second best solution. Based on the implementation, the solution in the append mode needs about half the time as the random write mode. This can be motivated by the fact that the number

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

of cache misses is lower in the append mode, than in the random mode. This characteristic increases with decreasing selectivity (0.05 and above), because the probability of the next TID being close enough to the previous TID and the corresponding memory segment (the bit) being already in the cache increases.

Compressed bitvectors behave worse. The reason for random access is that with every insertion of a TID, the compressed bitvector must be reorganized, which often has an influence up to the end of the whole compressed bitvector. This behavior occurs in the append and random modes for the WAH implementation (the WAH implementation has no special append mode, but only a *setBit(uint pos, bool value)* method to set a bit at an arbitrary position). However, the append mode could be implemented in a much more efficient way. The basic concept for the algorithm is represented in Figure 12. The idea of this implementation is that in the append mode only the last two words (LL: Last Literal, LF: Last Fill) must be considered: The last but one word, which is a literal, and the last, which is a 0-fill. Either the TID sets a bit in the last literal word, or the last fill must be split into two fills, with a literal in between (with holds the TID).

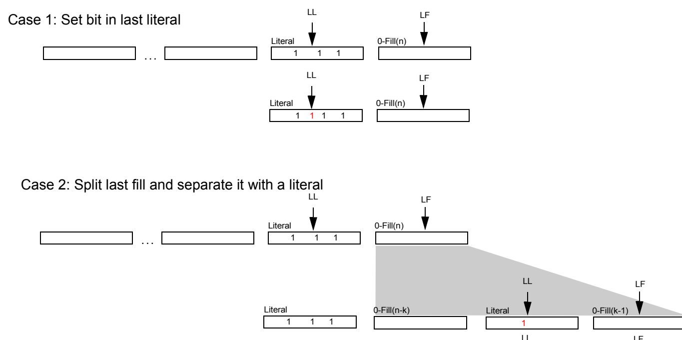

Figure 12. Appending TIDs in a compressed bitvector

To study writing of TIDs in the append mode, we extended the WAH library by the functionality described above. Figure 13 compares the time behavior for the uncompressed plain bitvector, the uncompressed WAH implementation, and our append-optimized implementation of the compressed append mode. The behavior of the append-only optimization is quite promising. It is about 25% better than using the uncompressed bitvector. The dynamic array still is the fast implementation, but, as we saw in Figure 7, memory consumption is highest, if the selectivity is low (high density values).

5) AND *operations on* Position Lists*:* Next, we perform an experiment to measure the time for *AND* operations. This is one of the basic operations performing the “WHERE” part of a query on a column store, where two or more *Position Lists* are *AND*ed (same with *OR*).

Figure 14 shows the results for the *AND* operation. As you can see, the time for *AND*’ing two uncompressed

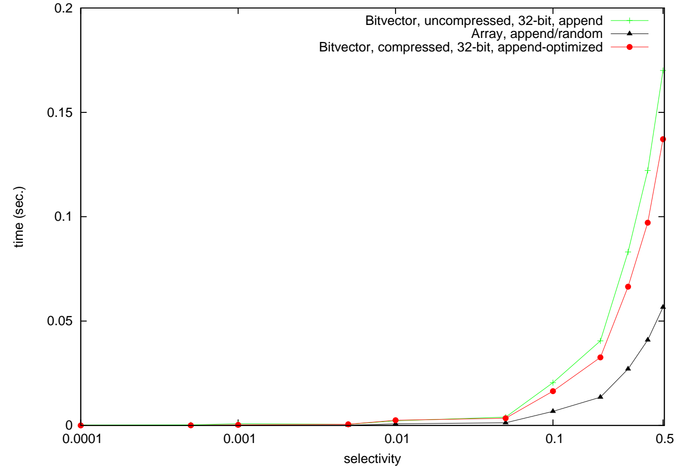

Figure 13. Comparision of appending TIDs to compressed and uncompressed bitvectors

bitvectors (both the plain bitvector implementation and the uncompressed WAH bitvector) is mostly independent of the selectivity. This can be understood, because the length of the vector is also independent of the selectivity and so the *AND* operation consists of a constant number of *and* instructions in the CPU. Comparing the uncompressed WAH bitvector and the plain bitvector, we are a little surprised. A slight overhead of the WAH implementation can be explained by the more complex algorithm and the additional memory consumption of 1/32 compared to the plain uncompressed bitvector. But our results show a significiant difference of more than 100% time penalty for the uncompressed WAH bitvector.

Also interesting are the results for the compressed bitvector and the array. While the array performs best for selectivities of 0.02 and higher, it degrades for lower selectivities (0.3 sec. for a density of 0.5). This is a little surprising, because the array implementation was one of the fastest in the previous experiments (iterating and writing TIDs). The degeneration can be explained by the caching strategies of modern CPUs. In the case of low selectivities, the two arrays grow and there is a cut-throat competition for places in the processor cache, which is why many cache misses result.

The compressed bitvector outperforms the uncompressed version for high selectivities (0.007 and above) because of its more compact representation and the ability to skip all the fill words completely. With lower selectivities, the fills get shorter and disappear later on. Hence, there is no advantage compared to the uncompressed representation. In this situation, the more complex algorithm is another drawback and leads to more instruction cache misses compared to the uncompressed version.

Again, the behavior for the array implementation is the most sensitive one. While the runtime behavior is the best for high selectivities, it completely degrades for lower se-

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

lectivities (density 0.03 and more).

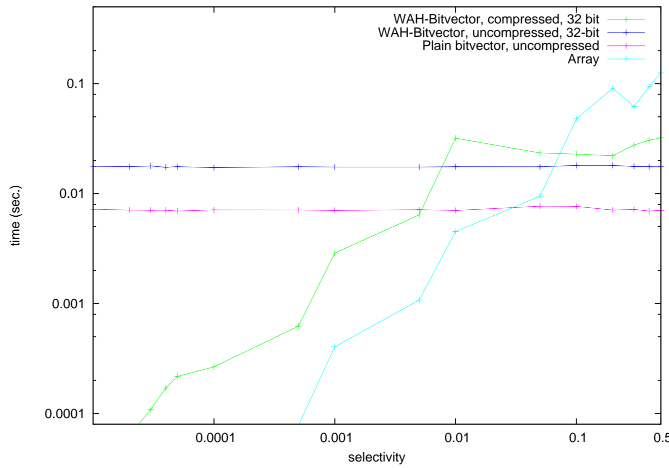

Figure 14. Measured time for *AND*ing two *Position Lists* with different implementations

6) Predicate Evaluation: In this experiment, we use *Position Lists* with different implementations (compressed bitvector, plain bitvector, array) and densities and compare their runtime behavior in evaluating the following expression:

```
column_1 = <value_1>
        and
column_2 = <value_2>
        and
column_3 = <value_3>
        and
column_4 = <value_4>
```

Each column contains 100 million datasets. In a first experiment, we formulate predicates that have the same selectivity for all four columns. In this case, the order of the evaluation of the predicates is irrelevant. We repeat this experiment with predicates of different selectivity. Figure 15 shows the execution time for the different implementations with eight different densities from 0.0001 to 0.5 (consider that the y-axis is logarithmic). The first observation is, that for densities of 1% and above the implementation of a *Position List* based on an array is about two orders of magnitude slower compared to the bitvector implementations. Also, it can be seen that for these densities, the uncompressed bitvector is the best implementation.

Additionally, we can see that the uncompressed bitvector has the same runtime behavior for all different densities. This can be explained easily by the fact that the uncompressed bitvectors have the same length for all densities and the same operations to perform. Especially for low selectivities (densities between 0.5 and 0.01), this is the preferred solution (also from the memory consumption point as shown in Figure 7). From a selectivity of 0.1%, the two other implementations perform better.

For selectivities greater than 1%, the compressed bitvector has an inferior performance compared to the uncompressed bitvector and the array implementation. This can be explained by the fact that the compression algorithm still compresses the bitvector, but only with a small compression ratio compared to the uncompressed version. The additional time results from the more complex operations while performing the *AND* operation. Starting with a density of 0.1% and lower, the compressed bitvector is the superior implementation. With a density of 0.01%, the compressed version is one order of magnitude faster than the array implementation and two orders of magnitude faster as compared to the uncompressed bitvector.

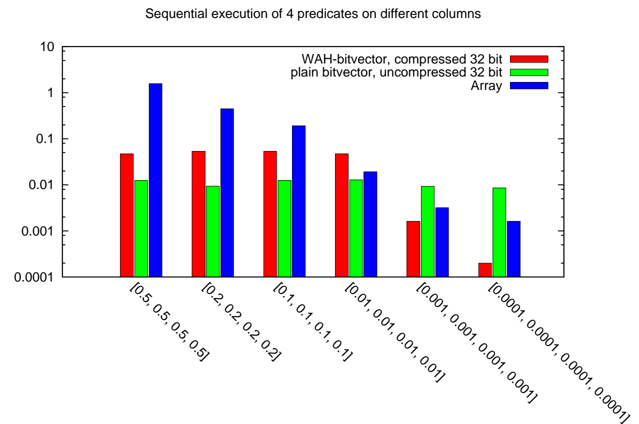

Figure 15. Measured time for the execution of 4 predicates on different columns using different implementaions for the *Position Lists*

So, as a rule of thumb, it can be said:

- If no information about the selectivity of a predicate is given, choosing an uncompressed bitvector for the implementation of the *Position List* is the best choice.
- If the density is expected to be 0.01 or higher, also use the uncompressed bitvector implementation.
- If the density is expected to be 0.01 or lower, use the compressed bitvector.

In the next experiment, we choose different selectivities for the predicates of the query. Again, we run multiple tests with different sets of selectivity. In this experiment, we vary the order in which the *Position Lists* are used to evaluate the expression. As expected, the runtime for the uncompressed bitvector is nearly independent of the density and the order of the *Position Lists*. Also expected, the runtime behavior for low selectivity queries (high density values) is bad for the array implementation (first two histogram groups). Arrays are also sensitive to the order of the *Position Lists*. The runtime behavior of the order [0.1, 0.2, 0.3, 0.4] is more

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

than twice as fast than the order [0.4, 0.3, 0.2, 0.1] (0.45 sec. vs. 1.0 sec.). This qualitative behavior holds for all densities. Interestingly, the behavior is inverse for densities greater than or equal to 0.1 in the case of the compressed bit vector. The reason for this behavior can be found in the implementation of the WAH algorithm. If no compression can be achieved, WAH switches automatically to the uncompressed representation. And as we have seen in Figure 14, using an uncompressed version is a little bit faster than using the compresssed version. For all lower densities, starting with the lowest density is faster. The runtime difference between the different orders degrades with lower densities (high selectivity). The last two histogram blocks also show that in the case of a first low density column, the order of the following columns is no longer critical. Nevertheless, the runtime behavior in these two last cases is determined by the low selectivity of the successive *Position Lists* and not by the first high selctivity *Position List*. If this would have been the case, the behavior of the third block (also starting with a density of 0.0001) should be expected.

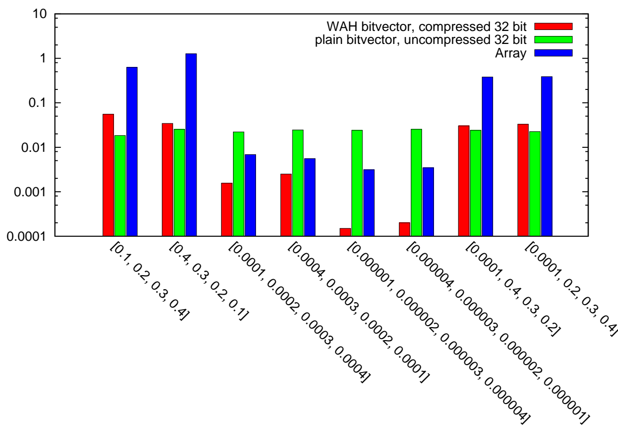

Figure 16. Measured time for the execution of 4 predicates on different columns using different implementaions for the *Position Lists*

B. Transformation between Different Representation Forms

Due to the strong influence of the selectivity, different representation forms of a *Position List* inside a complex query can be beneficial. This leads to the question how fast transformations between different representation forms can be performed. Therefore, we implement a number of transformation algorithms

($compressed \rightarrow array$, $array \rightarrow uncompressed$, $uncompressed \rightarrow array$, $array \rightarrow compressed$) and run a number of experiments, measuring the time of a *Position List* to change its internal representation. The tests are performed with different selectivities ranging from 0.0001 to 0.5. Figure 17 shows the results of these tests.

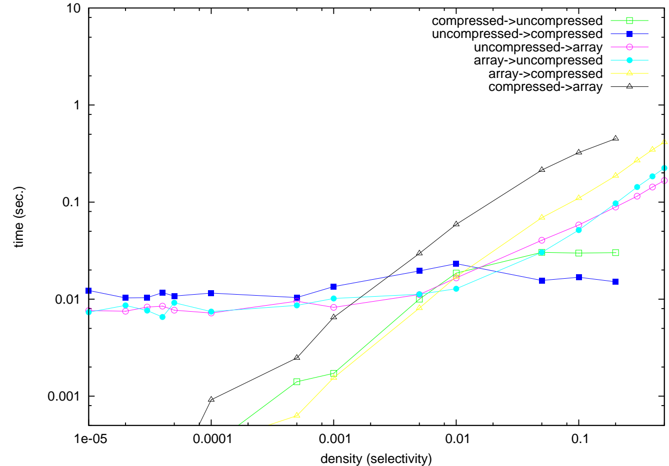

Figure 17. Time to transform different representation forms for *Position Lists*

Every line in the figure represents a concrete transformation. The x-axis represents the different densities and the y-axis the time for a transformation. Each transformation is done on a 100 million dataset *Position List*. One basic, but not surprising finding is that the transformation is faster for lower density values (high selectivity). This is based on the fact, that less TIDs must be transformed. In this case, the overall transformation time is determined by the memory footprint of the *Position List* representation, which is much smaller for the compressed representation (see Figure 7). As the transformation rules are more complex for the compressed representation, the growth in time with rising density is higher compared to the uncompressed representations (uncompressed bitvector and array). For densities greater than 0.2, WAH typically cannot compress the bitvector, because it needs at least 62 consecutive ’0’ values to achieve a compression. This is the reason why the transformations $compressed \rightarrow <x>$ and $<x> \rightarrow compressed$ are not shown for densities greater than 0.2. The transformation to a compressed bitvector is implemented using the append-optimized algorithm from Figure 12.

Based on the results shown in Figure 14, where the array was the fastest implementation for an *AND* operation with high selectivity *Position Lists* (density lower than 0.02), but also the worst for densites greater than 0.035, we can conclude:

- For small densities (< 0.005), where the array and the compressed bitvector are the favorable implementations with respect to runtime behavior (*AND*/*OR* operations), the transformation from an uncompressed bitvector has nearly constant cost for all densities and is determined by the memory footprint of the uncompressed bitvector. Nevertheless, the transformation time is the same as when performing a single *AND* operation with two uncompressed bitvectors.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

- For higher densities (> 0.01), where the cost of performing *AND* operations is up to two orders of magnitude higher when using an array implementation compared to the uncompressed bitvector, the transformation often makes sense (transformation time between 0.001 and 0.1 sec.).

VI. Conclusion and Future Work

The choice of the right data structure and algorithm for implementing *Position Lists* is not an easy task. It largely depends on the selectivity of the predicates and the operations to perform. Especially for low selectivities, the choice of the right solution is critical as was shown by the experiments.

The data structure of an array of unsigned integer values is outperformed by the uncompressed bitvector implementations by up to two orders of magnitude for low selectivities. On the other hand, it is a very good choice at high selectivities.

Uncompressed bitvectors have a predictable behavior for all selectivities, but are again outperformed by compressed bitvectors and arrays for very high selectivities.

If no information about the expected selectivity is available, using an uncompressed bitvector probably is a good choice. Depending on the selectivity and the used algorithm, the execution time is about three orders of magnitude and the uncompressed bitvector is of moderate performance.

Next, we will integrate the different implementations into our Column Store Toolkit (CSTK) [3] and perform experiments using the different implementations together with our toolkit components to measure the time behavior of our components with more complex queries like those from the TPC-H [24] benchmark.

Another interesting point is the implementation of *AND*/*OR* operations which allow *Position Lists* with different datastructures (i.e., array, compressed bitvector) as input (and output). These results can then be compared with the execution time of the version with an explicit transformation step before.

It has to be kept in mind that the ultimate goal is the development of a query optimizer for a *column store* [4].

References

[1] A. Schmidt and D. Kimmig, “Considerations about implementation variants for position lists,” in *DBKDA’13: Proceedings of the Fourth International Conference on Advances in Databases, Knowledge, and Data Applications*. IARIA, 2013, pp. 108–115.

[2] P. A. Boncz, M. Zukowski, and N. Nes, “Monetdb/x100: Hyper-pipelining query execution,” in *CIDR*, 2005, pp. 225–237.

[3] A. Schmidt and D. Kimmig, “Basic components for building column store-based applications,” in *DBKDA’12: Procceedings of the Fourth International Conference on Advances in Databases, Knowledge, and Data Applications*. IARIA, 2012, pp. 140–146.

[4] A. Schmidt, D. Kimmig, and R. Hofmann, “A first step towards a query optimizer for column-stores,” 2012, Poster presented at the Fourth International Conference on Advances in Databases, Knowledge, and Data Applications.

[5] I. Karasalo and P. Svensson, “An overview of cantor - a new system for data analysis,” in *Proceedings of the Second International Workshop on Statistical Database Management, Los Altos, California*, R. Hammond and J. L. McCarthy, Eds. Lawrence Berkeley Laboratory, 1983, pp. 315–324.

[6] G. P. Copeland and S. N. Khoshafian, “A decomposition storage model,” *SIGMOD Rec.*, vol. 14, no. 4, pp. 268–279, May 1985. [Online]. Available: http://doi.acm.org/10.1145/971699.318923

[7] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O’Neil, P. O’Neil, A. Rasin, N. Tran, and S. Zdonik, “C-store: a column-oriented dbms,” in *VLDB ’05: Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005, pp. 553–564.

[8] P. A. Boncz, M. L. Kersten, and S. Manegold, “Breaking the memory wall in monetdb,” *Commun. ACM*, vol. 51, no. 12, pp. 77–85, 2008.

[9] J. A. Khan and A. P. Shiralkar, “Article: Infobright enterprise edition analytic data warehouse technology ? an overview,” *IJCA Proceedings on National Conference on Innovative Paradigms in Engineering and Technology (NCIPET 2012)*, vol. ncipet, no. 15, pp. –, March 2012, published by Foundation of Computer Science, New York, USA.

[10] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, “Sap hana database: Data management for modern business applications,” *SIGMOD Rec.*, vol. 40, no. 4, pp. 45–51, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2094114.2094126

[11] R. MacNicol and B. French, “Sybase iq multiplex - designed for analytics,” in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, ser. VLDB ’04. VLDB Endowment, 2004, pp. 1227–1230. [Online]. Available: http://dl.acm.org/citation.cfm?id=1316689.1316798

[12] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear, “The vertica analytic database: C-store 7 years later,” *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1790–1801, Aug. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2367502.2367518

[13] “Oracle Database In-Memory Option — Feature — Oracle,” 2013, [Online; accessed 5-May-2014]. [Online]. Available: http://www.oracle.com/us/corporate/features/database-in-memory-option/index.html

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[14] "Columnstore Indexes for Fast Data Warehouse Query Processing in SQL Server 11.0," 2010, [Online; accessed 5-May-2014]. [Online]. Available: download.microsoft.com/download/8/C/1/8C1CE06B-DE2F-40D1-9C5C-3EE521C25CE9/Columnstore Indexes for Fast DW QP SQL Server 11.pdf

[15] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. row-stores: How different are they really," in *SIGMOD*, 2008.

[16] N. Bruno, "Teaching an old elephant new tricks," in *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA*. www.crdrdb.org, 2009.

[17] D. J. Abadi, D. S. Myers, D. J. Dewitt, and S. R. Madden, "Materialization strategies in a column-oriented dbms," in *Proc. of ICDE*, 2007, pp. 466–475.

[18] D. J. Abadi, S. R. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *SIGMOD*, Chicago, IL, USA, 2006, pp. 671–682.

[19] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden, "The design and implementation of modern column-oriented database systems," *Foundations and Trends in Databases*, vol. 5, no. 3, pp. 197–280, 2013.

[20] T. M. Chilimbi, B. Davidson, and J. R. Larus, "Cache-conscious structure definition," in *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1999, pp. 13–24.

[21] S. Manegold, P. A. Boncz, and M. L. Kersten, "Optimizing database architecture for the new bottleneck: memory access," *The VLDB Journal*, vol. 9, no. 3, pp. 231–246, 2000.

[22] M. Nelson, *The Data Compression Book*. New York, NY, USA: Henry Holt and Co., Inc., 1991.

[23] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing bitmap indices with efficient compression," *ACM Trans. Database Syst.*, vol. 31, no. 1, pp. 1–38, 2006.

[24] "TPC Benchmark H Standard Specification, Revision 2.1.0," Transaction Processing Performance Council, Tech. Rep., 2002.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org