# Considerations about Implementation Variants for Position Lists

Andreas Schmidt*† and Daniel Kimmig†
* *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}@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 an 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. 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

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 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 are 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 [1]. 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 preferrred 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 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 List*s. 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.

 ISBN: 978-1-61208-247-9 

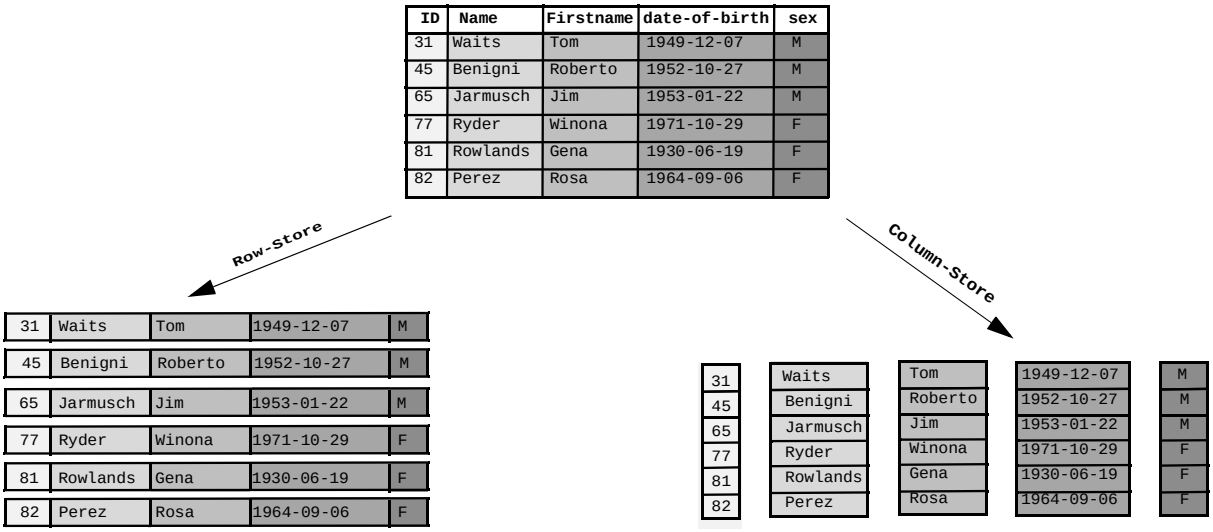


Figure 1. Comparison of the layouts of a row store and a column store (from [2])

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.

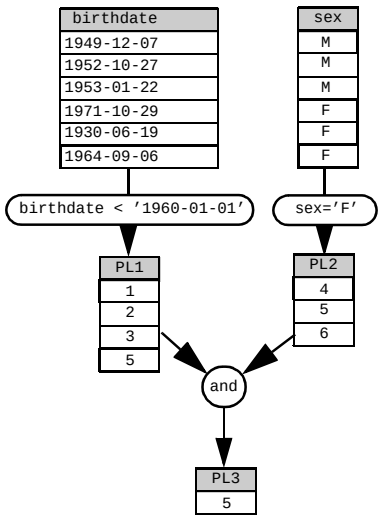


Figure 2. Isolated evaluation of predicates on their corresponding *Position Lists* and subsequent merging of the resulting *Position Lists* (from [2])

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.

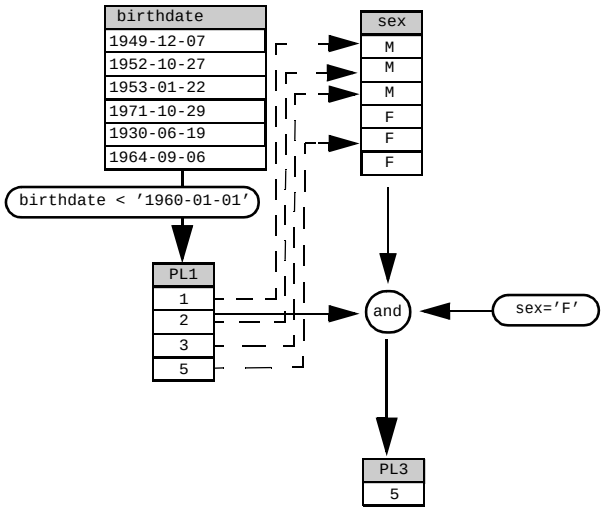


Figure 3. Iterative evaluation of predicates, using *Position Lists* as additional parameters

In previous work, we developed the Column Store Toolkit (CSTK) [2] and also used this toolkit as a starting point for further research in the field of optimizing SQL queries, based on a column store architecture [3].

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

 ISBN: 978-1-61208-247-9 

in different situations in terms of runtime behavior and memory consumption.

The paper is structured as follows: 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 the main part. Finally, results will be summarized, and an outlook will be given on future research activities.

## II. Related Work

Various publications compare the performance of column stores with that of row stores for different workloads [4], [5], [6]. In contrast to this, [7] examines different execution plan variants for column stores, while [8] considers the impact of compression. Following the work in [7], we examine different implementation variants for the underlying data structures and algorithms of the operations used in the execution plan of a query.

## III. Position Lists

From a logical point of view, a *Position List* is nothing else 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. With other words, a *Position List* stores the information for each tuple no matter whether if 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 [7] 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 [9], [10] 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*. *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 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 (density) 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).

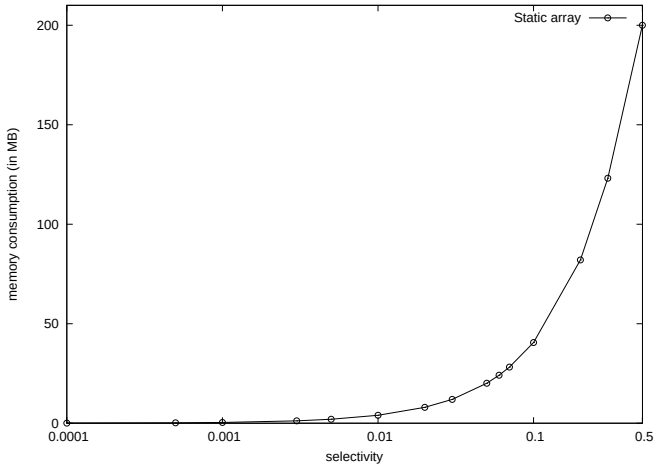


Figure 4. Memory consumption of a *Position List* implemented as array (logarithmic scale on x-axis)

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.

 ISBN: 978-1-61208-247-9 

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) [11]. 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.

The *Word Aligned Hybrid*-algorithm (WAH) [12] 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 bitmaps.

## 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 5 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 [2].

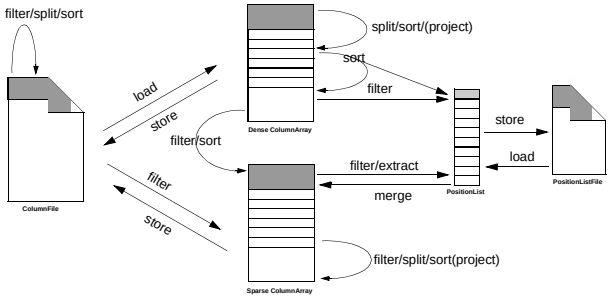


Figure 5. CSTK: Components and Operations (from [2])

## V. Measurements

### A. Elemental Operations

*1) Memory consumption:* In a first experiment we compare the size of the different data structures with respect to memory consumption. As shown in Figure 6, the behavior of the dynamic array implemetation is quite good for very small selectivities, but changes for the worse for medium and high densities. Uncompressed bitmaps (plain bitvector, WAH-uncompressed) behave independently for all densities, their size is determined by the number of tuples in a table only. Compressed bitmaps show a very good behavior for all densities. If the selectivities get low, they behave like uncompressed bitmaps (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.

*2) 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 by 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.

 ISBN: 978-1-61208-247-9 

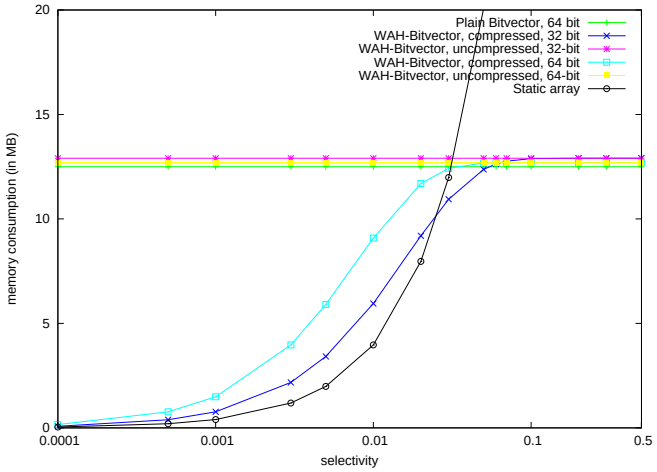


Figure 6. Memory consumption of different implementation alternatives for *Position Lists*

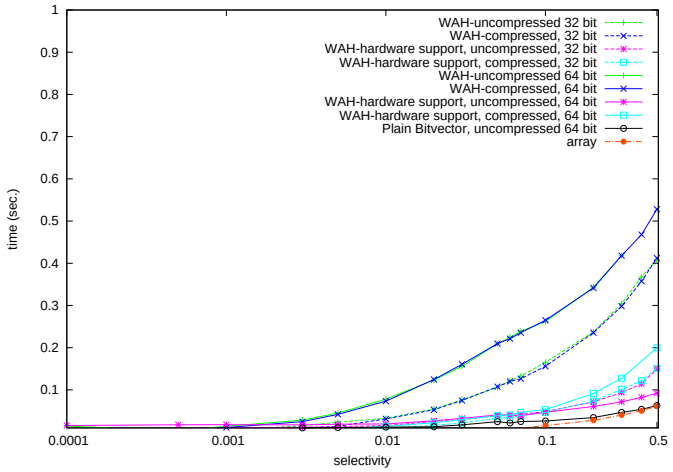


Figure 7. Measured time to iterate over 100 million data sets with different densities

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 measured the time to iterate over all the stored TIDs.

Figure 7 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 from 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 was the fastest).

Two more detailed graphs are given in Figure 8 and Figure 9. Here the static array implementation and the LZCNT supported iterators are considered for high and low selectivity, respectively.

While Figure 8 shows the details for selectivities between 0.0001 and 0.05, Figure 9 shows the lower selectivities between 0.05 and 0.5. One interesting point is, that with low selectivity (Figure 9) 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 founded 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.

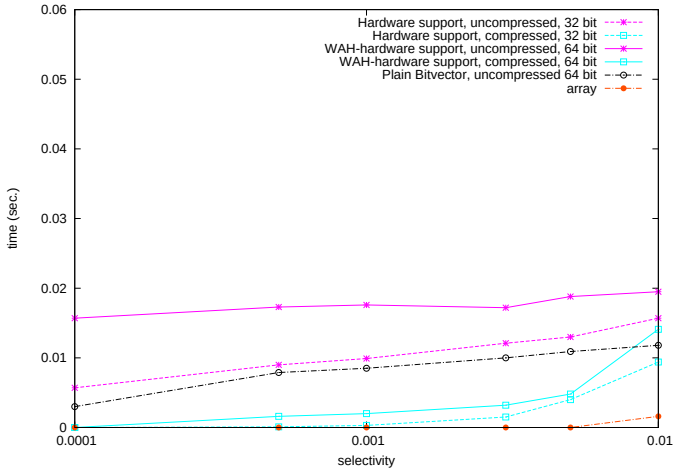


Figure 8. Measured time to iterate over 100 million data sets with high selectivity

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 6.

 ISBN: 978-1-61208-247-9 

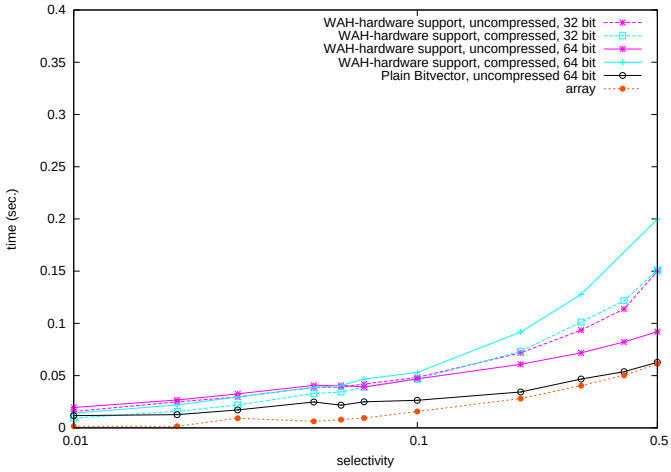


Figure 9. Measured time to iterate over 100 million data sets with low selectivity

*3) Writing TIDs:* In our next experiment we analyse 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".

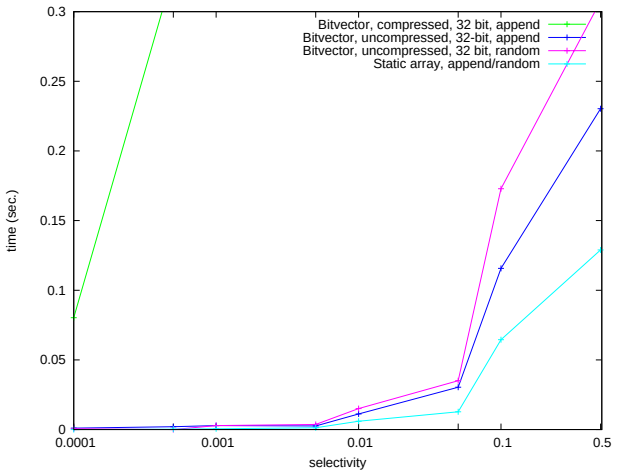


Figure 10. Measured time to write TIDs in different implementation variants for *Position Lists*

As a basic condition we can assume that writing of TIDs is mostly done in 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 10. 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 append mode is slightly faster than the random write mode. This can be motivated by the fact that the number 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 so 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 mode 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 11. 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 splitted into two fills, with a literal in between (with holds the TID). From the time behavior, we expect that this solution has about half of the performance of the uncompressed version (where we only have to jump to the corresponding word and set the bit), but is by far better than the general purpose *setBit* method from the WAH implementation.

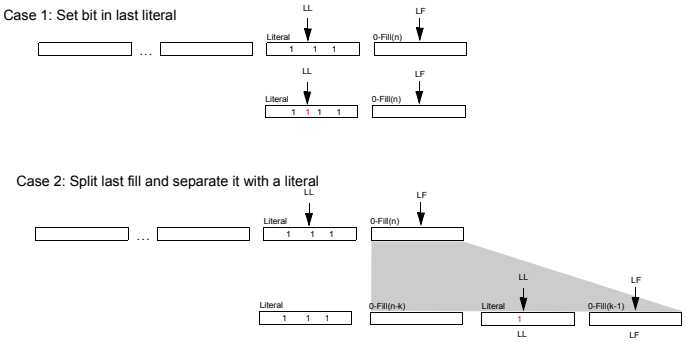


Figure 11. Appending TIDs in a compressed bitvector

*4)* 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 12 shows the results for the *AND* operation. As you can see, the time for *AND*ding two uncompressed bitvectors (both, the plain bitvector implementation and the uncompressed WAH bitvector) is mostly independent of the selectivity. This can easily be understood, because the length

 ISBN: 978-1-61208-247-9

of the vector is also independent of the selectivity and so the the *AND* operation consists of a constant number of *and* instructions in the CPU. The 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.

The more interesting lines are the compressed bitvector and the array. While the array performs best for selectivities of 0.02 and higher, it degrades for lower selectivities. 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.

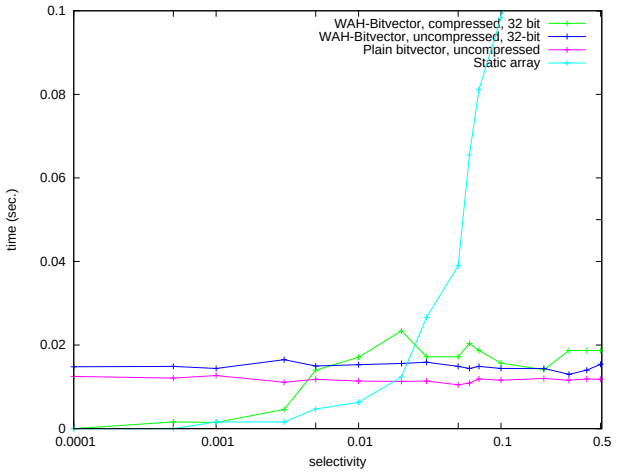


Figure 12. Measured time for *AND*ing two *Position Lists* with different implementations

## 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 a dynamic 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 very good choice at high selectivities.

Uncompressed bitvectors have a predictable behavior for all selectivities, but are again outperformed by compressed bitmaps 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 ranges over three orders of magnitude and the uncompressed bitvector is of moderate performance.

Next, the "append-mode" will be used for setting bits in a compressed bitvector. With this implementation. we will be able to perform more experiments with repect to the runtime behavior of complex conditions in both the sequential (Figure 3) and parallel (Figure 2) mode. After finishing this task, we will perform some more experiments using the different implementations together with our toolkit components to measure the time behavior of ouf our components with more complex queries like those from the TPC-H [13] benchmark.

Another interesting point is the usage of different data structures in one query. This may sound strange, but the conversion of compressed into uncompressed bitvectors and vice versa is very fast compared to the penalty of using the wrong algorithm/data structure. After evaluating the first predicates using uncompressed bitmaps (which perform well for low selectivities), the overall selectivity will increase and use compressed bitvectors could be advantageous.

It has to be kept in mind that the ultimate goal is the development of a query optimizer for a *column store* [3].

## References

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

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

[3] A. Schmidt, D. Kimmig, and R. Hofmann, "A first step towards a query optimizer for column-stores," in *DBKDA'12: Procceedings of the The Fourth International Conference on Advances in Databases, Knowledge, and Data Applications*. iaria, 2012.

[4] 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.

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

 ISBN: 978-1-61208-247-9 

[6] 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.

[7] D. J. Abadi, D. S. Myers, D. J. Dewitt, and S. R. Madden, "Materialization strategies in a column-oriented dbms," in *In Proc. of ICDE*, 2007.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

 ISBN: 978-1-61208-247-9 