Speeding Up Queries in Column Stores
A Case for Compression

Christian Lemke[1,2], Kai-Uwe Sattler[2], Franz Faerber[1], and Alexander Zeier[3]

[1] SAP AG, Walldorf, Germany
c.lemke@sap.com, franz.faerber@sap.com
[2] Ilmenau Univ. of Technology, Ilmenau, Germany
kus@tu-ilmenau.de
[3] Hasso-Plattner-Institute, Potsdam, Germany
alexander.zeier@hpi.uni-potsdam.de

Abstract. BI accelerator solutions like the SAP NetWeaver database engine TREX achieve high performance when processing complex analytic queries in large data warehouses. They do so with a combination of column-oriented data organization, memory-based processing, and a scalable multiserver architecture. The use of data compression techniques further reduces both memory consumption and processing time. In this paper we study query operators like scan and aggregation on compressed data structures implemented in TREX.

1 Introduction

Recent years have seen growing demands on data warehousing and OLAP technologies being able to handle terabytes of data and complex analytic queries from several hundred users simultaneously. Furthermore, more and more customers need ad-hoc and realtime evaluation of queries that make materialization of results and precalculation of reports more or less useless.

In order to meet these requirements, the limiting factor of disk I/O in database systems has to be addressed. Basically, in the area of data warehousing there are currently three main approaches: *(1)* Reduce the amount of data to be read from disk and read it as fast as possible. Besides index structures, column-oriented data organization shows great potential. *(2)* Avoid disk access completely by keeping and processing data in memory. *(3)* Exploit the computing capacity of a large number of inexpensive servers by using parallel processing.

In the field of Business Intelligence (BI) technologies some or all of these approaches are currently combined in the concept of BI accelerator solutions or analytical engines. An example is the SAP NetWeaver BWA based on TREX. TREX runs in a scalable multiserver architecture on blade servers. Processing is performed completely in main memory, fact and dimension tables are organized column-wise (vertically partitioned) and the columns are partitioned horizontally among the server nodes. The combination of parallelism and main memory processing allows interactive execution of analytical queries without pre-aggregation.

T.B. Pedersen, M.K. Mohania, and A M. Tjoa (Eds.): DaWaK 2010, LNCS 6263, pp. 117–129, 2010.
© Springer-Verlag Berlin Heidelberg 2010

However, since RAM is still expensive compared to hard disk drives and cannot be enlarged indefinitely, very large data warehouse installations require a large number of server nodes. Furthermore, scanning the entire memory of a node does not allow to exploit the benefit of L2 caches. Therefore, data compression techniques are used to reduce the data volume and hence improve the cache utilization. But, compressing data only is not the silver bullet: If data processing as part of query evaluation requires an expensive decompression or additional memory space, the benefit of compression is mitigated or even lost.

Thus, query operators should process compressed data directly without computing-intensive decompression. In our work, we investigate such strategies for implementing query operators. Based on the TREX infrastructure and the discussion of several compression schemes for in-memory column stores we present strategies for filters, scans and aggregations which exploit these compressed data structures. The results from our experimental evaluation show that depending on the data distribution significant performance improvements can be achieved.

2 Related Work

The first work that suggests working directly on compressed data as long as possible was [4]. Its focus is on joins but also exact match comparisons for selections and duplicate elimination (grouping) is described. They use domain coding as lightweight compression in a row store implementation and specify the needed properties of compression techniques for an efficient query processing. The decompression of values cannot be avoided for most aggregation functions and when the data has to be displayed to the user. To reduce the repeated decompression of fields [10] introduce an extended iterator model.

Raman and Swart showed in [7] that fast processing is also possible on heavyly compressed data in a row store. They present a novel Huffman coding scheme to evaluate equality and range predicates on compressed data without full dictionary access. Before working on compressed data can take place, the compressed records and fields have to be extracted. Afterwards index scan, hash join, merge join, grouping and count (distinct) aggregation can work directly on the Huffman codes. For the min and max computation the codes of each code length need to be decompressed for comparison. This is because of the coding scheme where only the codes with the same length are ordered.

By contrast in [2,13] the data is always decompressed for query execution.

One of the first papers about working on compressed data in column stores was [3]. Here the scan and join operators in queries are executed in a main memory database on domain coded data. An additional speedup is gained by using multidimensional hash tables as indexes.

Based on the column-oriented C-Store [9] Abadi et al. [1] introduced an architecture for a query executor that works directly on compressed data. In contrast to this work, we use data structures that represent the data of a whole column and not only parts of it.

There is also research on avoiding decompression while querying for other database architectures. In [5] linearized multidimensional arrays are used to efficiently aggregate data whereas [6] concentrate on joins in a binary relational database where triples are stored.

Another approach to speed up the query execution on compressed data structures is the use of modern hardware. [12] as an early paper uses SIMD instructions in a column store to exploit parallelism and eliminate branch mispredictions. Zhou and Ross consider sequential scan, aggregation, index operations and joins. For a single compressed tuple instead of several values of one field [8] evaluates a conjunction of equality and range predicates using SIMD.

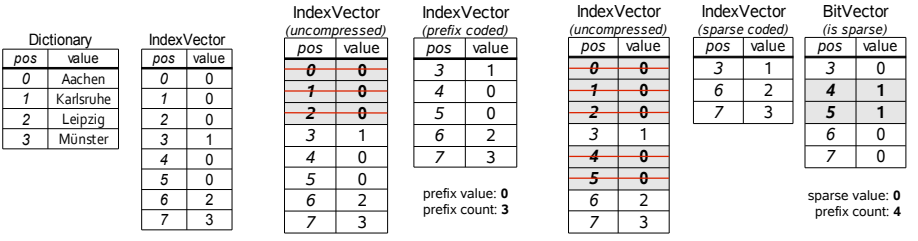
3 Data Structures for Column Compression

In the following we briefly describe some of the data structures used in TREX for storing compressed column values. These data structures are pure main memory structures and the main goals are *(1)* an effective compression scheme to reduce the memory consumption and *(2)* allow to process queries without decompressing the data. Particularly, the latter goal requires efficient access both to individual values and also to blocks of values.

The basis for all the following techniques is *domain coding* [10,1,13]. For domain coding, all values from a column are stored in lexicographical order in a dictionary. The original column is replaced by an index vector that stores only bit-compressed pointers to the dictionary. To minimize the bit lengths, a total of u distinct values appearing in n rows are coded using $n\lceil \log_2 u \rceil$ bits. Figure 1(a) shows an example of domain coding for the sample data *Aachen, Aachen, Aachen, Karlsruhe, Aachen, Aachen, Leipzig, Münster*, where each value is represented in the index vector by two bits. Note, that the column `pos` is given in the figures only for illustrative purposes and not physically stored. The use of integers instead of the original values has the advantages that it reduces the data volume to be processed and allows to exploit hardware optimization for integer processing. Furthermore, multiple values can be read at once into the CPU cache and processed in parallel with special SIMD processor commands.

Based on this scheme *prefix coding* can be applied as a simple compression technique. Here, repetitions of the same value at the start (prefix p) of a column are deleted and replaced by one value and its frequency. For a column with n elements and u_{col} distinct values, $(n-p)\lceil \log_2 u_{col} \rceil + 64$ bits are required. Fig. 1(b) shows an example of the original uncompressed index vector and its compressed version that is constructed using prefix coding.

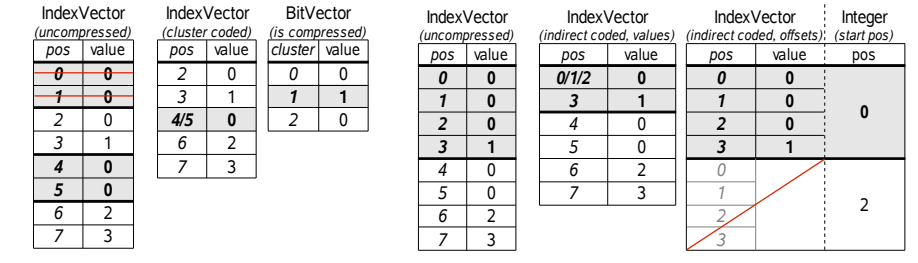
If the most frequent value appears not only in the prefix but also scattered among the other values, then *sparse coding* can be applied to achieve a good compression (Fig. 1(c)). Here the positions of all appearances f of the most frequent value are recorded in an additional bit vector and the original values or deleted from the index vector. It is possible to use prefix coding for the bit vector, which for a large prefix p can further improve the compression ratio. With this technique only $(n-f)\lceil \log_2 u_{col} \rceil + (n-p) + 64$ bits are needed.



(a) Domain coding (b) Prefix coding (c) Sparse coding

Fig. 1. Examples of domain, prefix and sparse coding

All the techniques described in the following will use prefix coding and work on blocks of data containing minimal numbers of distinct values in order to achieve a good compression ratio. In *cluster coding* only blocks with a single distinct value are compressed by storing only the single value. Additionally, a bit vector is needed to indicate which blocks are compressed, in order to be able to reconstruct the original column. In this paper, we do not further consider how to determine the optimal block size and its impact on the compression rate and query runtime. In any case the number of elements should be an integral power of two, so that instead of multiplication and modulo operation we can exploit fast bit operations. Figure 2(a) shows an example in which the block size is two and the compressed values are shown in gray boxes.



(a) Cluster coding (block size: 2) (b) Indirect coding (block size: 4)

Fig. 2. Examples of cluster and indirect coding

If the data blocks contain more than one but only few distinct values, *indirect coding* can be used. Here domain coding is applied to appropriate blocks, which adds the indirection of a separate mini-dictionary for each block. To reduce the number of dictionaries and hence the memory consumption, one dictionary can be used for a continuous sequence of blocks as long as any new entries in the dictionary do not increase the number of bits required to code the entries. For a column with u_{col} distinct values, a block with k values and u_{block} distinct values benefits from indirect coding if and only if the dictionary and the references take less space than the original (domain coded) data: $u_{block}\lceil \log_2 u_{col} \rceil + k\lceil \log_2 u_{block} \rceil < k\lceil \log_2 u_{col} \rceil$.

Figure 2(b) illustrates the data structures used in the implementation. Here one block contains four values and the compressed elements are shown again in gray boxes. The dictionaries and the uncompressed data are stored in the middle index vector, *values*, and each block is addressed with a start position. In the data structure on the right, compressed blocks have their own index vector, *offsets*, which points to the individual values.

Finally, a slightly modified variant of *run length coding* can be used, which compresses sequences of repeated values to a single value for each run together with the number of repetitions. In order to calculate the start position of each value in the run, we add up the frequencies of the previous values. However, this can result in a high overhead, so instead we decided to reduce the compression slightly by storing the start position and not the number of repeats.

To speed up single-value accesses we introduce two inverted index structures: a blocked and a signature index. The *blocked index* stores for each value a list of blocks in which this value occurs and is therefore only applicable if using cluster, indirect or run length coding. For the *signature index* the data is split into a fixed number of parts and for each value the occurrences in that parts are stored in a bit vector which significantly reduces the amount of storage.

4 Query Operators for Compressed Columns

In this section we will describe in detail how the compressed data structures are used in the standard query operators. We designed the compression techniques for an efficient direct access to the data and gaining significant performance improvements over uncompressed data structures especially for the scan operator. In the evaluation we will show that the choice of the best compression technique depends on the data and can be done automatically.

4.1 Basic Operators

The basic operator in a data warehouse environment is the *scan* operator with optional filter predicates. In scenarios with mass data and non selective predicates, the performance of the scan operator is very critical. The *get* operator provides random access to a column by retrieving the value (or a reference to the dictionary) of a given row id. This operation is needed for projections and selective filters. The performance of the operators depends on the result materialization data structures (like bit vector or integer vector), the used compression technique and many more factors. In the following we will present the scan and get operators for the most complex compression techniques: sparse and indirect coding.

Sparse Coding. As mentioned above, the sparse coding maintains a bit vector B_{nf} for the most frequent value v_{f} of a column. A bit is set if the value of the corresponding row is not v_{f}, otherwise the bit is unset. For all compression techniques a prefix offset p specifies the first row id, which is different from the prefix value v_{p} that is omitted. The values that are different from v_{f} are stored in an index vector I_{nf}.

Scan Operator. The scan starts by testing the predicate for v_p when there is a prefix p. The same is done for v_f whose corresponding row ids can easily be extracted from B_{nf}. If necessary, I_{nf} is also scanned and the corresponding row ids are calculated by counting the unset bits in B_{nf}:

1. estimate the current row id by assuming that no values are removed (row id = relative row id)
2. determine the number n_{nf} of non frequent values (unset bits)
3. repeat the following steps until n_{nf} is higher than the relative row id
   (a) increase the current row id
   (b) if the current value is not v_f then increase n_{nf}
4. calculate the absolute row id with current row id + p

In order to accelerate the calculation of unset bits a data structure is introduced, which stores the number of set bits for every block of rows (i.e. all 128 rows) as a sum of all previous blocks.

Get Operator. The get operator first checks if the requested value is in the prefix. If the row id is higher than p then B_{nf} is tested. If the corresponding bit is unset then the result is v_f, otherwise the position of the requested value in I_{nf} is calculated and the requested value is extracted from there.

The sparse coding shows very good performance characteristics if the sparse value is dominant in a way that it covers more than 90% of the rows. Otherwise the costs of the indirect access over the bit vector is too high as shown in the evaluation.

Indirect Coding. By neglecting the prefix, the indirect coding is a mixture of variable and fixed size coding. Per block of rows the references to a local block dictionary are of fixed size, but each block (e.g. 1024 rows) has its own dictionary which references to the global dictionary of the column. For blocks where local dictionaries are not suitable (e.g. with as many distinct values as rows) no local dictionary is used, but the references to the global dictionary are stored without further indirection. Because the references in the local dictionaries and the references in blocks without local dictionaries are of fixed size, they are stored in an index vector I_v. A second vector V_b stores information for each block, like the start position in I_v and the references to the local dictionary (if it exists).

Scan Operator. The scan operator scans I_v by testing the predicate for every value and storing the hit positions in a temporary structure. For each entry in this structure the following is done:

1. determine the corresponding block in V_b by checking the start position
2. if the block is uncompressed (no local dictionary) then calculate the absolute row id with (block number * block size) + match pos - start pos
3. if the block is compressed (local dictionary)
   (a) calculate the local dictionary reference with match pos - start pos
   (b) scan the local references for that reference
   (c) calculate the absolute row id from the local reference hit positions

Get Operator. Retrieving the value of a given row id starts by calculating the block number in V_b. If the block is uncompressed, the position in I_v is row id - (block number * block size) + start position. If the block is compressed, the position in the local references is row id - (block number * block size). The final position in I_v is the local reference plus the start position.

Hardware Optimization. Another approach to speed up data processing is the implementation of the scan and decompression operations using SIMD instructions. The SIMD implementation used in our work is extensively evaluated in [11] and we will only investigate the influence on our compressed data structures.

4.2 Aggregate Operators

Though, in this work we focus on the efficient implementation of scan operators because they are most critical regarding performance, we discuss in the following aggregation operators, too.

We start with a description of single column aggregations and grouping. These aggregation operators are implemented as part of the scans such that filter predicates can be calculated without additional efforts. Furthermore, some aggregate functions such as `min` and `max` can be evaluated using the dictionary only.

Because columns are partitioned horizontally among all servers, aggregate operators are evaluated in two steps: a scan phase that is performed in parallel on all partitions followed by a merge phase. During the scan phase, partial aggregates are computed for each partition which are then merged into the final aggregates. For sparse coding the scan phase is performed in the following way. We assume a single column grouping with `count` as aggregate function. In order to collect the counts per group an array G is used that is indexed by the values of the dictionary (i.e. the positions of the actual values in the dictionary). Let I denote the index vector storing the column values in sparse coding and B the corresponding bit vector.

1. Read the first value v_0 from the dictionary and set $G[v_0] = p$
2. Determine the number n of bits set in B and let $G[v_0] = G[v_0] + n$
3. Scan the index vector for all values v_i and let $G[v_i] = G[v_i] + 1$

After the aggregate array G_i of all partitions are calculated, they are merged in a straightforward way. This step is simplified by the same ordering of all grouping arrays because on all partitions the same dictionary is used.

For other compression schemes this approach has to be slightly modified. For example, for cluster coding the bit vector is scanned in parallel to the index vector. If a bit i is set, the corresponding entry i of the index vector is skipped and $G[v_0] += \textit{blocksize}$. Indirect coding requires an additional step to process the index vectors of the individual blocks.

This single-column aggregation scheme can be extended to the multi-column case by maintaining a single group array for all grouping columns $G[c_1, \ldots, c_m]$ and perform the scans on these columns in parallel. Then, for each tuple $v_1, \ldots, v_m$ the corresponding entry in G is updated as described above.

5 Experimental Evaluation

This section presents the results of evaluating the scan operator on different synthetic datasets without a query optimizer. We use a micro benchmark to analyze special properties of the compression techniques, which can be found also in real datasets. The data consists of one column with 10 million data items and 4472 unique integer values. One exception is the **single** dataset which contains only one unique value. In the **linear skew** distribution the value i occurs $i + 1$ times contiguously and in the **uniform** distribution all values occur equally often. For the **sparse** dataset the items are consecutively numbered values and a very frequent (sparse) value is added. The position of the sparse data items can be grouped at the top, at the bottom or evenly scattered. In the **blocked** data, blocks with one or more unique values are generated. We have 2236 single-value blocks and 2236 multi-value blocks with 447 unique values where the block size is for both cases 2237 values.

The results of our experiments are shown relative to the domain coded data (index vector) to exclude effects of different data types. Furthermore the cluster and indirect coding use a block size of 1024 values in our implementation. If not stated otherwise the used scan operator gets a value range as input and writes the results in a vector. Furthermore we only evaluated the blocked index of the presented inverted index structures.

All experiments were performed on an Intel® Xeon® processor X5650 with 2.67 GHz. The scalability of the index vector if using more cores has already been shown in [11]. So because the memory bandwidth required by the compressed data is lower, we concentrated on single-core measurements.

5.1 Experiments without SSE (Streaming SIMD Extensions)

First we want to show that with an optimal compression, not only the memory size is reduced but also queries can be accelerated. The dictionary coded **single** dataset needs with 1 bit per value around 1221 KiB. Because the presented compression techniques omit the prefix they need less than a kibibyte to store the values. Even in sparse coding the bit vector will not be stored.

Figure 3 overviews the memory consumption of the distributions we focused on in our evaluation. The **linear skew** distribution in Fig. 3(b) shows that if the most frequent value is below a certain threshold the additional costs of the bit vector in the sparse coding leads to an increased memory size. Because a value occurs in less 1024-blocks, the additional memory used for the inverted indexes is low. The run length coding can adapt best to the different-sized single-value blocks which leads to significant reduction. Using the **uniform** distribution the cluster coding performs better because of the bigger single-value blocks. For the other techniques there are only small differences in the memory size.

If the most frequent value in the **sparse** dataset is evenly scattered (Fig. 3(d)) the sparse coding reduces memory size most and only cluster and indirect coding without inverted indexes can exploit the block of the most frequent value at the end. This single-value block arises from the scattered generation and a sparse

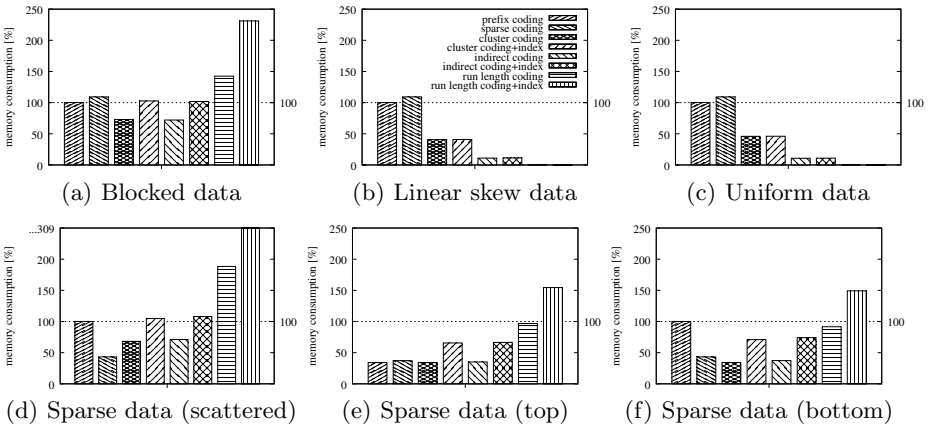


Fig. 3. Memory consumption

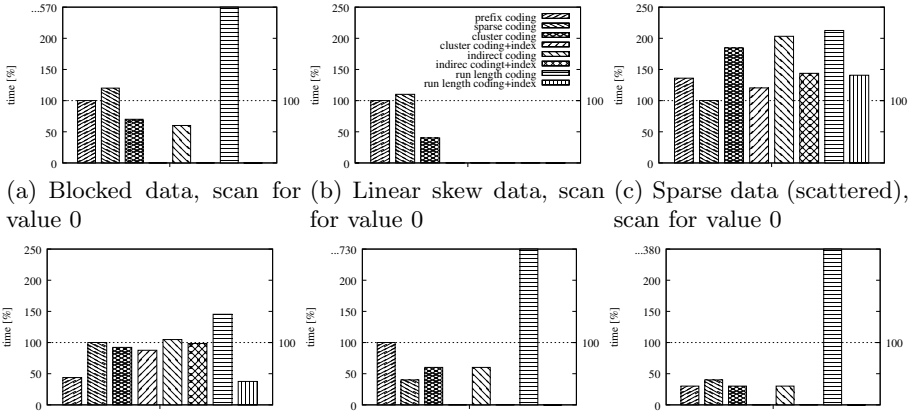
occurrence of more than 50% (in our case 66%). Because of the many value changes the run length coding performs worst. For better readability we omit a part of the bar graph and specify the maximum value on the y-axis.

If the most frequent value is at the top (Fig. 3(e)) all compression techniques apply prefix coding which leads to a significant memory reduction. Sparse coding needs slightly more memory because of the additional bit vector. Because of many value changes and less big single-value blocks in the **blocked** dataset (Fig. 3(a)) only cluster and indirect coding can reduce the memory consumption.

To show the possible savings in execution time we do a scan for value 0 and value 1 in Fig. 4. Value 0 is the most frequent value in the sparse dataset. In Fig. 4(a) with the blocked dataset the speed of the query correlates to the size of the data structures except when using inverted indexes. Here the total size of the index structure is big but only the blocks specified in the inverted index need to be scanned which results in a very fast execution. For the linear skew dataset Fig. 4(b) shows that of all blocked compression techniques only cluster coding cannot adapt well to the small single-value blocks and that is why it performs worse. If the most frequent value is scattered in the sparse dataset (Fig. 4(c)) it is the worst case distribution. Here just the sparse coding can compete with a query on the only dictionary coded data because of the reduced amount of data to scan. In Fig. 4(d) one can see clearly that less memory consumption does not always imply a faster execution. Also the row id reconstruction overhead has to be taken into account like in the case of applying sparse coding.

If the result size is small like in the scans showed in Fig. 4(f) and 4(e) the row id reconstruction and result vector resize costs are negligible compared to the savings. Because for value 1 there are no single-value blocks, the very frequent single-value access to two index vectors is too expensive. That is also the case if the amount of data to scan is less (Fig. 4(f)).

All measurements show that when using inverted indexes scan times for single values always decrease. Run length coding is only worth if there are few but big single-value blocks and the fasted technique if using an inverted index. Cluster



(a) Blocked data, scan for value 0

(b) Linear skew data, scan for value 0

(c) Sparse data (scattered), scan for value 0

(d) Sparse data (top), scan for value 0

(e) Sparse data (scattered), scan for value 1

(f) Sparse data (top), scan for value 1

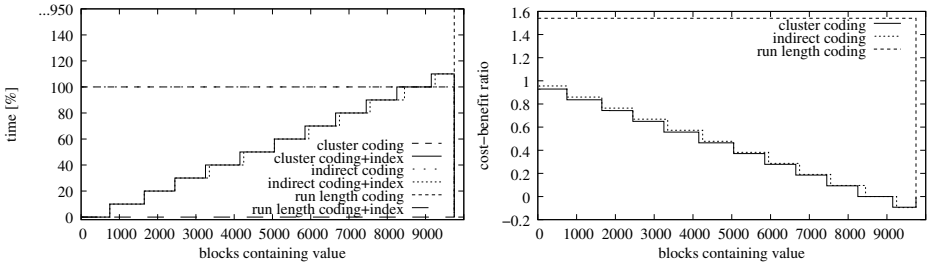
Fig. 4. Single-value scans

coding on the other hand has the best memory-speed tradeoff for all evaluated data distributions.

We further expect that using inverted indexes is only faster than the baseline if the requested values are located in few blocks. Figure 5(a) shows the times of the experiments where we varied in how many blocks value 0 occurs by using a step size of 100 blocks. The data distribution is based on the sparse dataset and value 0 is occurring 9766 times which corresponds to the number of blocks with a size of 1024 values. Because of the many value changes and the small amount of occurrences of the value the run length coding is constantly around ten times slower than the baseline and for this reason we cut the graph for better readability. If using the inverted indexes for the cluster and indirect coding the scans become slower the more blocks have to be considered.

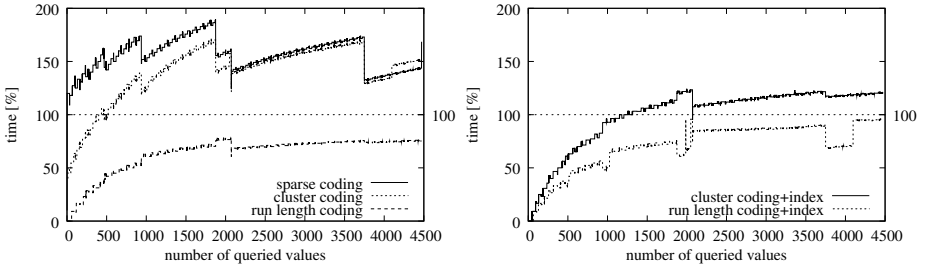
To determine if it is worth to spend more memory for the inverted index we calculate a cost-benefit ratio by dividing the time gained by the additional memory needed. Figure 5(b) shows that for run length coding the higher memory consumption always results in an increased speed. For the two other techniques it depends strongly on the distribution of the value and at some point the additional memory used is counter-productive. To deal with this problem in our implementation we do a full table scan if an indexed value occurs in too many blocks.

Next we want to show that the more values are queried the slower the query becomes. In this experiments we use the uniform data distribution and do a scan for 1, 2, ..., all values. The results are shown for the slowest, an average and the fastest compression technique with and without using an inverted index in Fig. 6. One can see that the scans using run length coding are always faster and sparse coding always slower than the baseline. The outliers in the graph result from copying memory when resizing the result vector using a doubling strategy.



(a) Time (b) Cost-benefit ratio

Fig. 5. Increasing number of blocks containing the requested value



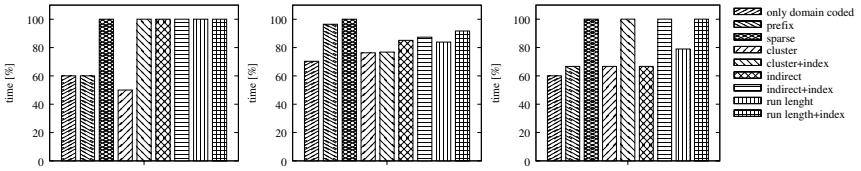
(a) Without inverted indexes (b) With inverted indexes

Fig. 6. Multi-value scans

Furthermore the dictionary coded algorithms use another initial vector size because of the missing prefix handling. The cluster coding is only worthwhile if the number of requested values is small because of the increasing row id reconstruction costs. Another reason for the increasing time needed if using a inverted index are the increased number of blocks to consider for the scan.

5.2 Experiments with SSE

Finally, we show in Fig. 7 the performance improvements possible when using SSE as SIMD implementation on the compressed data structures. The baseline is the scan times measured without active SSE. The results show that the biggest saving in time is achieved for the domain and prefix coded data because here always full table scans take place and SSE is optimized for mass data processing. If the result size is increasing as in Fig. 7(b) the performance gain is decreasing because of the reconstruction and resizing costs. There is no or just a little benefit by using SSE in combination with the inverted indexes because here the amount of data to be scanned is already significantly reduced. Also the sparse coding technique cannot exploit the advantages of SSE. Reasons are the reduced data size if the most frequent value occurs often (Fig. 7(b) and (c)) and that most of the time is spent for row id reconstruction (Fig. 7(a)). The times for cluster



(a) Linear skew data; value 0 (b) Sparse data (top); value 0 (c) Sparse data (top); value 1

Fig. 7. Single-value scans using SSE

coding when using the linear skew (and uniform) dataset are reduced as there are few compressed clusters which leads to more data that has to be scanned.

6 Conclusion

Data compression is an important technique to reduce memory consumption and query processing time in data warehouse systems. However, the real benefit of compression can be only leveraged if the decompression effort can be minimized. To tackle this problem, we have presented in this paper several dictionary-based compression schemes as well as query operator implementations for scans and aggregates which work directly on the compressed data. Our experimental results show that depending on the data characteristics and appropriate compression techniques significant improvements in query processing time can be achieved, but require a careful choice of the compression scheme. We partly implemented an optimizer that determines an optimal row order, the appropriate compression techniques and inverted index structures depending on the data distribution.

References

1. Abadi, D.J., Madden, S.R., Ferreira, M.C.: Integrating compression and execution in column-oriented database systems. In: Proc. SIGMOD, pp. 671–682 (2006)
2. Chen, Z., Gehrke, J., Korn, F.: Query optimization in compressed database systems. In: Proc. SIGMOD, pp. 271–282 (2001)
3. Cockshott, W.P., McGregor, D., Wilson, J.: High-performance operations using a compressed database architecture. The Computer Journal 41(5), 283–296 (1998)
4. Graefe, G., Shapiro, L.D.: Data compression and database performance. In: Proc. ACM/IEEE-CS Symp. on Applied Computing, pp. 22–27 (1991)
5. Li, J., Srivastava, J.: Efficient aggregation algorithms for compressed data warehouses. IEEE TKDE 14(3), 515–529 (2002)
6. O'Connell, S.J., Winterbottom, N.: Performing joins without decompression in a compressed database system. SIGMOD Rec. 32(1), 6–11 (2003)
7. Raman, V., Swart, G.: How to wring a table dry: Entropy compression of relations and querying of compressed relations. In: Proc. 32nd VLDB, pp. 858–869 (2006)

8. Raman, V., Swart, G., Qiao, L., Reiss, F., Dialani, V., Kossmann, D., Narang, I., Sidle, R.: Constant-time query processing. In: Proc. 24th ICDE, pp. 60–69 (2008)
9. Stonebraker, M., et al.: C-store: A column-oriented dbms. In: Proc. 31st VLDB, pp. 553–564 (2005)
10. Westmann, T., Kossmann, D., Helmer, S., Moerkotte, G.: The implementation and performance of compressed databases. SIGMOD Rec. 29(3), 55–67 (2000)
11. Willhalm, T., Popovici, N., Boshmaf, Y., Plattner, H., Zeier, A., Schaffner, J.: Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. PVLDB 2(1), 385–394 (2009)
12. Zhou, J., Ross, K.A.: Implementing database operations using simd instructions. In: Proc. SIGMOD, pp. 145–156 (2002)
13. Zukowski, M., Héman, S., Nes, N., Boncz, P.: Super-scalar ram-cpu cache compression. In: Proc. 22nd ICDE, p. 59 (2006)