

FCompress: A New Technique for Queriable Compression of Facts and Datacubes

Pedro Furtado, H. Madeira

CISUC/Depto Eng. Informática Univ. Coimbra

Pinhal Marrocos

3030 Coimbra, Portugal

pnf@dei.uc.pt

Abstract.

Decision support applications must analyze information from data warehouses efficiently. For this reason, huge data warehouses must have mechanisms to cope with massive amounts of data. Reducing and compressing fact tables, summary tables and data cubes is important for faster operation and smaller storage overhead. Traditional compression techniques are not useful in this context except for archiving, because they render the data unqueriable. Although data reduction techniques are useful for fast approximate answers to complex queries, their accuracy is not enough to replace the base data. In this paper we present FCompress, a new fact compression technique that effectively replaces the base data, compressing it while maintaining queriability. The approach is based on the premise that a very small and adjustable error is acceptable in many fact attributes. The technique is applicable to fact and summary tables and data cubes alike. It has been evaluated, showing that very small errors can be achieved for point reconstruction (typically below 2%) while the original fact table is reduced to about 35% to 60% of its size and the data cube is reduced to about 15% to 30% of the size. The error is even smaller for typical OLAP queries, usually less than 1%, depending on the degree of aggregation.

1. Introduction

Data Warehouses and OLAP tools must manage and analyze enormous amounts of data. These systems provide specific structures to manage several

dimensions of the business process from large amounts of raw transactional data that is periodically loaded from several sources. The major problem in those systems is the huge size of multidimensional fact-like data (fact tables of star schemas, summary tables that are created to pre-compute aggregations, data cubes and subcubes that are used in OLAP analysis). Fact and summary tables are the central elements of star schemas, implementing relational online analytical processing (ROLAP) in the decision support systems (DSS) and account for a large portion of the total space. A typical data warehouse is made of a federation of such facts. Summary tables or materialized views are used to speedup the processing of queries, specially the ones involving large aggregations of data and expensive joins of several tables. Data cubes, on the other hand, are often used directly or as internal (temporary) representations. Data cubes, sub-cubes and cuboids are computed and materialized from aggregation lattices to speedup typical OLAP analysis. They are alternative representations of multi-dimensional data sets and we include them in the discussion as fact-like representations.

As a consequence of the large storage space occupied by the fact-like data sets, there is also a performance degradation. Indexes and summary tables are used to improve significantly the performance for very large data sets. Unfortunately, they also have a significant effect on the increase of storage space. Indexes are indispensable, though, for efficient handling of very large tables. Summary tables or materialized views pre-compute important quantities, so that queries aggregating over large ranges can be answered faster. They also aggravate

the storage and management overhead significantly because they are large and multiple views are needed to deal with the different query patterns (these query patterns are dependent on the usage).

Data reduction techniques [4] have little impact on the storage overhead, reduce the size of the data set significantly and achieve a large query speedup. On the other hand, they do not achieve good accuracy for many important query patterns and their applicability is limited.

Normal compression techniques can achieve very large compression rates, but they are not usable in this context, except for data archiving. This is because the data must be compressed by blocks, loses all queriability, performance is severely degraded and it is not possible to search for patterns without incurring in expensive decompression of the blocks first.

There are a number of requirements for a successful fact compression strategy:

- maintain (or improve) queriability (SQL queries) – queriability and performance requirement;
- return either exact or very accurate results for any query – accuracy requirement;
- effectively compress the data - compression requirement;
- be integrated in the datawarehouse context – compatibility requirement;
- be applicable to any fact-like data set (fact and summary tables, data cubes)– scope requirement.

The new technique presented in this paper (FCompress) satisfies these requirements. It is based on the quantization [11, 16] of value attributes and stores compact bitcodes which are obtained using a lossy, non-uniform and highly adaptable quantizer.

The technique achieved a compression of tested fact tables to a value between 35 and 45% of the original size (with value attributes being compressed to between 15% and 30% of their size). In those data sets, individual point values can be reconstructed with negligible error (below 2%). Point reconstruction is essential when the goal is to replace the original data set (compression), and also to guarantee accurate results for all OLAP query patterns. It should be noted that traditional data reduction techniques can not reconstruct point values

with acceptable accuracy and often produce bad estimations of some group results in OLAP queries.

FCompress can be applied to any fact-like data set (facts, summary tables, data cubes). The dimension attributes of fact tables should not be discarded, as they play a major role in the structure of the data warehouse (this issue does not concern data cubes as they do not represent the dimension attributes explicitly). Accordingly, FCompress can organize the data as index-structured tables (whose entries can be compressed further), reducing the space occupied by the dimension attributes without loss of queriability.

The paper is organized as follows: section 2 briefly discusses related work on data compression and data reduction techniques. The compression of fact-like data sets is introduced in section 3. Section 4 describes the strategy used to compress measure attributes and the compression of dimension attributes is treated in section 5. Section 6 provides an exhaustive analysis and evaluation of the approach. Section 7 concludes the paper.

2. Related Work on Data Compression and Data Reduction

Traditional compression techniques are not useful in the data warehouse context. They achieve large compression rates that are very useful for archiving using algorithms such as arithmetic coding, run-length encoding, header compression, huffman coding, Lempel-Ziv or transform-based compression [12, 15, 16, 19, 20]. However, they are block compression techniques. In general, the compressed data sets are not directly queriable without prior decompression of whole blocks, due to the variable-length of the compressed fields. The term “variable-length field” refers to the use of a variable number of bits to represent compressed values. Compression and decompression are done by blocks and require expensive access methods to locate the correct block. Decompression requires bit processing operations and access to auxiliary structures such as dictionaries or code trees to decode the bitstream. Indirect access and decompression cause a large overhead. These observations are also true for typical compression packages such as zlib [6].

Typical data reduction algorithms include sampling [3, 7], singular value decomposition (SVD)

[4], wavelets [17, 18], regression [1, 2] and histogram-based techniques such as MHIST [14]. Most of these techniques produce a histogram with a set of regions (buckets), stored either as data cube chunks or table tuples. Bad estimations can result from queries that intersect small fractions of individual buckets (e.g. analyzing the sales of individual products over large periods of time may cut-through buckets in the product dimension because individual products are being selected). Point reconstruction, which is important in the compression context, is also not possible with normal data reduction techniques. Summaries made of randomly extracted samples, such as [7], are useful for statistical analysis of the data, but have similar accuracy limitations.

FCompress is a proposal of a compression technique that does not have the problems associated with the block compression characteristics of traditional compression and maintains all queriability and compatibility in the data warehouse context. Additionally, it does not have the accuracy and applicability limitations of data reduction techniques. The compressed data set can replace the original data set with negligible error.

3. Compression of Facts using FCompress

Consider the structure of facts and summary tables as $\text{fact}(d_1, \dots, d_n, v_1, \dots, v_m)$, where d_1, \dots, d_n are dimension attributes and v_1, \dots, v_m are value attributes. Figure 1 shows a star schema with dimension and fact tables. The symbols D and V are the dimension and value attributes respectively. A data cube is also shown with value attributes V.

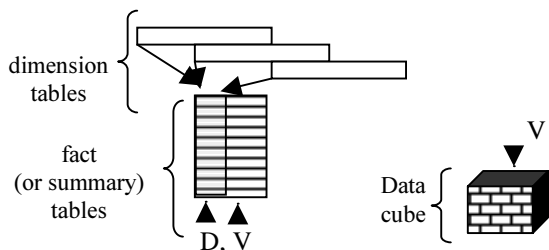


Figure 1 – Illustration of Fact Data Sets

The compression strategy used in FCompress replaces V attributes (v_1, \dots, v_m) of fact tuples or data cube cells by much smaller representations consisting of packed bitcodes. These bitcodes are obtained by

quantizing [11, 16] the value attributes. FCompress can also compress the dimension attributes by representing them in a compressed index-organization. Figure 2 shows the structure of the resulting compressed data sets.

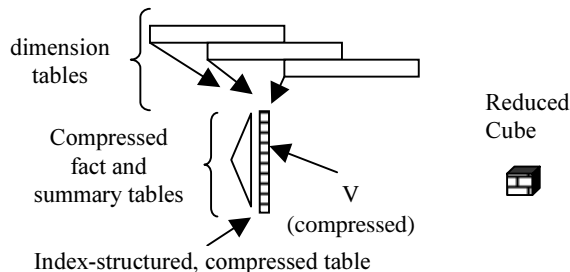


Figure 2 – Illustration of Compressed Data Sets

3.1 Rationale Behind the use of Adaptable Quantization

Quantization [11, 16] is used in FCompress to determine a set of representative values for measure attributes. The crucial issue of accuracy is guaranteed by our proposal for an “adaptable” quantization procedure, which uses an accuracy constraint based in user-defined error bounds during the determination of quantization levels. The number of levels is determined adaptively and attributes that cannot be estimated with large accuracy using a reasonable number of levels can be left uncompressed.

The specific data warehouse context is important, because many measure attributes offer favorable cases for the replacement by a number of representative values. The best results (most accurate and with less bits) are achieved when the data set is made of a small set of distinct values (e.g. 100M rows with 1000 distinct values), in which case the bitcodes are simply a much more compact representation of the values. Quantization also returns very good approximations when a significant fraction of the values are distributed around sets of clusters, in which case reconstruction values converge to the centers of distinct clusters. The importance of this property comes from the fact that typical data warehouse attributes measuring quantities such as daily or periodic sales are often distributed around clusters and repeated values. As more levels are added, the quantizer finds the most appropriate positions for the quantization levels to represent most values accurately.

Both the analysis and experimental results show that the approach guarantees accuracy and typically provides almost exact value reconstruction and precise query answers. This is an important issue, as the technique maintains the queriability of uncompressed data but is also able to guarantee high levels of accuracy for every query pattern and even in the reconstruction of individual values.

4. Compression of Measure Attributes

The most important phase of FCompress is the compression of the value attributes using the following steps:

- Value attributes are analyzed to determine a fixed-width compressed coding that is optimal with respect to a set of error and space criteria. This is achieved with adaptable quantization.
- The tuples or data cube cells are coded and packed to achieve a large compression rate.
- Extremely fast decompression of values “on-the-fly” is achieved by unpacking and dequantizing the values with very fast low-level operations.

The following steps and items are involved:

- **Determination of Representative Values** – this step determines enough representative values such that the values taken by any attribute (or attributes) can be represented exactly or with a totally adjustable amount of error. This is done using the adaptable quantization strategy.
- **Reconstruction Array** –representative values are chosen to be a power of two (2^n), so that they fit in exactly n bits. They are ordered by magnitude in a simple array (the reconstruction array), used to translate between values and bitcodes. The bitcodes coincide with the indices of the array.
- **Compression** – this step replaces the values by the bitcodes using the reconstruction array and packing (concatenating) those bitcodes to produce smaller tuples or data cube cells.
- **Decompression** – this step involves unpacking the bitcodes (deconcatenation) and retrieving reconstruction values from the array.

4.1. Adaptable Determination of Representative Values

The definition of quantization is sufficiently generic to incorporate four important sub-strategies:

- Lossless coding of the minimum number of distinct values of an attribute;
- Uniform quantization representing each value by the closest of k levels distributed with equal intervals in the domain of the values;
- Non-uniform quantization represents each value by the closest of k levels distributed such that the error in the estimation of values is minimized;
- A hybrid representation allows even the less favorable data sets to be quantized without error or with very small error, regardless of their characteristics. This strategy quantizes all repeated or clustered values into a set of representative values and leaves the rest of the data set uncompressed.

FCompress typically applies non-uniform or hybrid adaptable quantization by default, as this results in the best approximations and compression rates (the only advantage of uniform quantization in this context is extremely fast determination of the levels). A set of user-defined constraints on the error must be defined to guide the iterative quantization algorithm (which runs until they are met or a limit on the number of levels or time to converge is reached). Any subset of a number of constraints can be specified. The main constraints are the average or standard error, the maximum error, the fraction of outliers that is admitted and the percentiles of the errors. Hybrid compression also requires the specification of maximum thresholds on the distance of values to the closest quantization level (tuples with larger distances are left uncompressed).

Quantization is done using a randomly extracted training set, which should be large enough to include representative values from most relevant clusters in the data set (this issue is detailed in [4]). The algorithm is able to determine quantization levels efficiently for reasonably large training set sizes (up to 100K values were used). The output of the algorithm is the set of reconstruction values that are

stored in a small reconstruction list array associated with a data set.

The adaptability of the strategy is ensured by iterating over the number of quantizer levels, as shown in Figure 3.

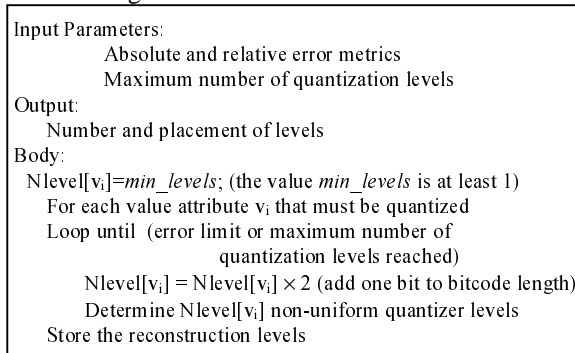


Figure 3 – Adaptable Quantization Algorithm

The core of the quantization procedure uses the well-known Max-Lloyd non-uniform quantization algorithm [11]. The implementation was optimized by minimizing the number of accesses needed per iteration and using a variance metric in the convergence of quantization levels. The algorithm is not discussed here for lack of space. A full discussion is available in [4].

4.2. Reconstruction Array

FCompress uses a small reconstruction array and a single direct access to a cell for decompression. The reconstruction array illustrated in Figure 4 is ordered by reconstruction values and has typical size between 64 and 8192 cells (corresponding to bitcode lengths between 6 and 13). Bitcodes coincide with the index of the cell. The array is accessed to retrieve the bitcode (index in the array) or the reconstructed value (value in each position of the array).

| | | | | | |
|-------|------|-------|-------|-----|-------|
| 3.234 | 9.21 | 14.76 | 21.37 | ... | 102.5 |
|-------|------|-------|-------|-----|-------|

Figure 4 – Illustration of a Reconstruction List

During compression, the reconstruction array is accessed to translate values into the bitcodes of the nearest reconstruction values (bitcode values coincide with the indices, as mentioned above).

During decompression, the bitcodes are translated back to the corresponding reconstruction values.

4.3. Compression

Compression requires an access to the reconstruction array to retrieve the bitcode that must replace the value and a concatenation of that value into a word. Each word contains a small number of packed bitcodes and is sized to minimize the overhead of decompression.

The lookup in the reconstruction array requires a binary search of the array to determine the quantization level. The search must return the index of the reconstruction value that is closest to the value that must be compressed:

Bitcode = bin_search(Reconstruction_List, Value);

This bitcode is then packed (concatenated) into the word using two low-level operations in the microprocessor register: one bitwise shift (left) and one bitwise OR operation per value. The operation is (C++ syntax, shift = <<, OR=|, AND=&):

Bitcode_i = ((data_register <<shift_i) | maskregister_i);

The rightmost bitcode requires only a masking instruction:

Bitcode_n = (data_register | maskregister_n).

4.4. Decompression

The most crucial operation is decompression, as it is used for “on-the-fly” reconstruction in query processing. It requires unpacking the bitcodes and a direct access to the reconstruction array, which is conveniently small to fit in the microprocessor cache after a few initial misses.

Unpacking (deconcatenation) requires only a bitwise shift (right) and an AND instruction in a microprocessor register for each value that must be decompressed to answer the query:

Bitcode_i = ((data_register >>shift_i) & maskregister_i)

The rightmost bitcode requires only a masking instruction:

Bitcode₀ = (data_register & maskregister₀);

The reconstruction value is retrieved using a direct access to the reconstruction list:

Reconstructed_value= Reconstruction_List[bitcode];

Choice of a Convenient Word Size: the overhead of the low-level unpacking operation is minimized when an integer number of packed bitcodes is put into a “convenient” unit of storage with a size of one (or a few) microprocessor register words. This way each word contains an integer number of bitcodes, which are located in the same offsets in consecutive words and do not “overflow” between words. This minimizes the operations required and the shifts are always done in a single instruction.

If the integer number of bitcodes resulting from the quantization procedure does not fit exactly in that word, some bits are wasted (decreasing the compression rate). However, the number of bits wasted is usually only a small fraction of the size of the word because the word is large (typically one or two processor words). Additionally, “wasted” bits can be used to improve the accuracy by adding quantization levels to attributes that are not quantized to exact values.

5. Compacting Dimension Attributes

While data cubes are highly compressed directly using this strategy, the amount of compression of fact tables is limited by the existence of dimension attributes that cannot be compressed, to maintain the compatibility with other data warehouse and OLAP structures. However, the representation can be changed to index-organized tables for larger compression rates. Index-organized tables exist in database engines such as Oracle [21]. These tables have the data rows grouped according to the primary key. This clustering is achieved using a B*tree index, which is a special type of index tree that differs from regular table B-tree indexes in that it stores both the primary key and non-key columns. The attributes of index-organized tables are stored entirely within the physical data structures for the index and full-table scans return all rows in primary key order. The entries of the index can be compressed further, although that possibility is not explored here.

6. Analysis and Evaluation of the Technique

6.1 Analysis of the Compression Rate

Typically, data cubes are compressed 2 to 5 times by coding and packing. For relational data, the compression rate is given by:

$$\frac{\text{n}^\circ \text{ of index blocks} + \text{n}^\circ \text{ of table blocks (dims + values)}}{\text{n}^\circ \text{ of index blocks} + \text{n}^\circ \text{ of table blocks (concat. bitcodes)}}$$

In order to estimate the compression rate analytically, the size of the index and the table must be estimated first. Both are stored in database blocks and therefore depend on a number of parameters with example values given in Figure 5 (Oracle).

| Parameter | Description | Values |
|-----------------|-------------------------|--------|
| block_sz | block size | 8192 |
| block_hd_sz | block header | 90 |
| ndx_block_hd_sz | index block header | 160 |
| col_sz | avg row length table | 32 |
| ndx_col_sz | avg row length index | 16 |
| col_n | Number of cols table | 8 |
| ndx_col_n | Number of columns index | 4 |
| pctfree | % of avail. space index | 2 |

Figure 5 – Parameters Used in the Analysis of Compression Rate

The approximate number of index blocks needed for the primary index and table are given by:

$$\frac{\text{n}^\circ \text{ of index blocks} = \text{n}^\circ \text{ of rows} / (100 - \text{pctfree}) \times (\text{block_sz} - \text{ndx_block_hd_sz})}{100 \times (\text{ndx_col_sz} + 8 + \text{ndx_col_n})}$$

$$\frac{\text{n}^\circ \text{ of table blocks} = \text{n}^\circ \text{ of rows} / (100 - \text{pctfree}) \times (\text{block_sz} - \text{block_hd_sz})}{100 \times (\text{col_sz} + 3 + \text{col_n})}$$

The space occupied by each row in the table or entry in the index is given by adding the values,

- The number of bytes used for each row or entry;
- Assume 3 bytes for row overhead, 8 bytes for index entry overhead (Oracle);

- The number of columns in the table or entry, as one byte is needed to indicate the length.

These formulas were used to determine the compression rates of Figure 6. The results assume that attributes have an average of 4 bytes and half or a quarter of them are value attributes, depending on the total number of attributes being less or more than 10, respectively. Quantization was assumed to compress value attributes to 25% and 15%. “Index” refers to the compression produced by organizing the fact as an index-organized table. “FCompress” refers to the compression rate when only the measure attributes are compressed. Finally, the compression rate of the quantized, index-organized data set is given by “FCompress+Index”.

The results show that FCompress+Index achieves a compression rate between 1.75 and 2.25.

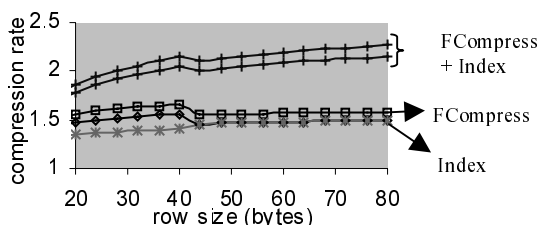


Figure 6 – Compression Rates of FCompress

6.2 Performance Analysis

In this section, we show experimentally that the decompression overhead is practically insignificant by comparing the time required to decompress and operate a compressed data cube in memory to the time required to operate directly on the uncompressed data set. This guarantees acceptable performance in any query. Additionally, a large speedup is achieved in queries that need to scan significant portions of the data set, because a much smaller number of blocks must be retrieved from disk. This property was also verified experimentally.

Speed of Decompression (for Compressed Data Cubes in Main Memory): Consider an operation such as aggregation (possibly grouping by categories) that must process values over large ranges into one or more groups. Every value from the range has to be loaded into the microprocessor and added to a partial sum. Assuming that the compression rate is

$cmprss_rate$, only a fraction $1/cmprss_rate$ of the memory accesses is required to load the same number of values of the compressed data into the microprocessor (e.g. one fourth if the compression rate is 4 times). On the other hand, the operation on compressed data incurs in the extra overhead of a number of unpack and lookup operations equal to the number of values in the data set. Assuming that Y values are operated upon in the query, the operations that are necessary are described next (we use the example $Y = 100$ million values with 4 bytes each – 400MB - and a compression rate of 4),

- Uncompressed data cube:
 - 4Y bytes loaded from memory
(100 M values = **400 MB**)
 - + Y × aggregation-related operations
(100 M operations)
- Compressed data cube:
 - Y bytes loaded from memory
(100 M values = **100 MB**)
 - + Y unpack operations (100 M ops)
 - + Y lookup operations (100 M ops)
 - + Y × aggregation-related operations (100 M ops)

Operation on the compressed data cube accesses a much smaller number of memory words, although the bitcodes must be unpacked and looked up. Predicting which of these operations is fastest is not trivial. It depends on the scheduling of instructions, the time taken to execute them and the use of internal caches by the microprocessor in the lookup functionality. However, by compiling and executing the code in two different systems, we have concluded that the overhead incurred by decompression is smaller than the overhead required to fetch much more data from memory for uncompressed data (an extra 300MB in the example!). The extreme speed of the unpack operation and the fact that the lookup operation hits the same small number of cells in a microprocessor cache repeatedly were also decisive in this important result.

Experimental Results: This experiment shows that the “on-the-fly” decompression of values on compressed data cubes does not introduce major performance degradation, with or without code optimizations. In fact, operation on the uncompressed data set could only match the efficiency of operation on the compressed data set when loop unrolling was used [10] (decreasing the overhead of the instructions

controlling the main cycle). Figure 7 shows the CPU time taken to execute a simple total sum on the compressed (Cmprss) and uncompressed (UnCmprss) data sets (similar results were obtained in varied aggregations). The execution time using loop unrolling is also shown (UnCmprss, Unroll). The compressed data set has to be unpacked and looked up, while operation on the uncompressed data set must load more values (the compression rate of the experiment was 4 times). The results are shown with and without the code optimization options of Visual C++ turned on (Maximize Speed). This code was executed in a Pentium II-300Mhz processor (similar comparative results were achieved in a Unix workstation with a totally different processor).

These results essentially show that “on-the-fly” decompression is extremely fast and incurs in practically insignificant overhead.

| Size(MB) Normal /Cmprss | Cmprss | UnCmprss | UnCmprss, Unrolled |
|-------------------------------|--------|----------|-----------------------|
| 25/6.25 | 0.601 | 1.352 | 0.610 |
| 50/12.5 | 1.132 | 2.744 | 1.192 |
| 100/25 | 2.353 | 5.498 | 2.354 |
| 150/37.5 | 3.455 | 8.232 | 3.555 |
| 200/50 | 4.607 | 11.026 | 4.736 |

Figure 7 – Performance of Sum Operation

Scanning the Data Set from Disk: Consider the size of the data set M , the size of the disk block B and the compression rate $cmprss_rate$. The simplest operation, scanning the whole data set from disk, requires M/B I/O operations. The number of I/O operations on the compressed data set is approximately $1/cmprss_rate \times M/B$. As I/O operations are orders of magnitude slower than memory accesses and decompression involves insignificant overheads, the operation on compressed data is much faster. Figure 8 shows the results of an experiment corresponding to a full scan of a data set of tuples from disk to sum the values. The data sets with sizes between 100MB and 800MB were compressed to half of that size (compression rate = 2). As expected, the operation is much faster on the compressed data set, even though the value attributes had to be decompressed (I/O was implemented with normal Visual C++ routines).

| Size(MB) normal/cmprss | Cmprss | UnCmprss | UnCmprss Unrolled |
|---------------------------|--------|----------|----------------------|
| 100/50 | 26.4 | 49.431 | 46.287 |
| 200/100 | 57.2 | 104.01 | 97.72 |
| 400/200 | 127.6 | 212.158 | 205.868 |
| 600/300 | 170.3 | 333.631 | 322.87 |
| 800/400 | 284.2 | 495.07 | 473.548 |

Figure 8 – Time to Scan and aggregate the Data Set from Disk (secs)

Data cubes can be compressed even further (up to 4 times), achieving larger speedups in scanning operations.

6.3 Analysis of the Accuracy

The accuracy of the quantization procedure is very good, due to the adaptable number of levels. Most typical OLAP measure attributes are quantized with very small error and some attributes with no error at all. Furthermore, typical OLAP aggregation queries are answered with completely insignificant errors.

The non-uniform quantizer determined using the Max-Lloyd algorithm tries to minimize the square error, causing quantization levels to converge to the places that achieve such minimization. The worst accuracy happens when the data set has a completely uniform distribution. The resulting quantizer is uniform (the quantization levels are located at regular intervals). For a data set with size S and N quantization levels, the error can be measured using the variance, which is given by:

$$s^2 = 2 \times \frac{1}{S/N} \int_0^{S/2N} x^2 dx = \dots = 1/12 (S/N)^2$$

This error is directly proportional to the square of the size of the data set (S) and inversely proportional to the square of the number of quantization levels (N). On the other extreme, the best case for the quantizer accuracy happens when the attribute takes only N distinct values (regardless of the number of values and size of the domain). In this case, the variance is null and the compression results lossless. If the maximum number of quantization levels is N_{max} , data sets with up to N_{max} distinct values are quantized without loss of information (e.g. up to 4096 values if the quantizer uses 12 bits).

The most frequent case is in between the two previous extremes. This is modeled by considering that the values of the data set fall into three categories after they are quantized:

- Fraction of values f_{cl} forming clusters (modeled by a normal distribution around the center). Clusters are quantized by at least one level (placed at the centroid of the cluster) or more levels placed around the centroid in such way that the variance due to the cluster is minimized;
- Fraction f_0 of values repeated several times. One level is required for each repeated value;
- Fraction f_u of values distributed in an approximately uniform way;

Quantization levels converge to positions that minimize the error, which is a consequence of the Max-Lloyd iterative algorithm. This way, clustered values are approximated by one or a few levels until the variance (s_{clnvl}^2) is minimized. Repeated values are approximated using a single level with no error and the remaining values are considered uniformly distributed. In [5], the variance of a uniformly distributed fraction of the data set f_u was determined as $f_u^2 \times 1/12 (S/N)^2$ by considering the discrete version of the formula shown before. The overall variance is given approximately as the sum of the contributions from the constituent fractions,

$$s_q^2 = f_0 \times 0 + f_{cl} \times s_{clnvl}^2 + f_u \times s_u^2$$

$$\approx f_{cl} \times s_{clnvl}^2 + f_u^3 \times 1/12 (S/N)^2$$

The adaptability given by varying the number of levels means that levels are assigned until the clusters and repeated values have a very small variance. As a result, the only significant term that cannot be completely handled by quantization is the one corresponding to the uniformly distributed fraction of the data set (f_u). As long as that fraction is much smaller than one or the size of the data set S is not too large in comparison to the number of levels N , the quantizer will have very good accuracy. Otherwise, either larger errors are accepted or the attribute is left uncompressed.

The adaptable nature of our algorithm guarantees that additional bits are added to the bitcode length (doubling the number of levels) until the error constraints specified by the user are met.

Figure 9 and Figure 10 show the average, maximum and standard deviation of the (value-relative) error as the number of bits used in the quantization procedure increases (each additional bit doubles the number of quantization levels). Figure 9 is an example of an attribute which is easy to compress with a reasonably small number of bits (7 or 8 bits are enough). The attribute of Figure 10 was more difficult to quantize. It required at least 10 bits (1024 levels), although 11 bits would produce better estimations (2048 levels). We have used up to 13 bits (8192 levels) in practical applications.

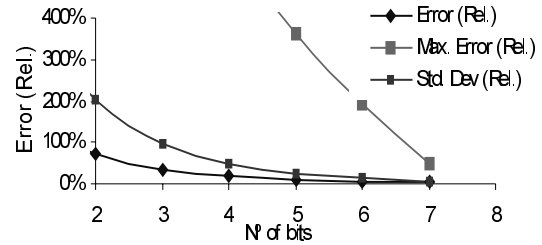


Figure 9 – N° of levels vs Accuracy for Regular Attribute

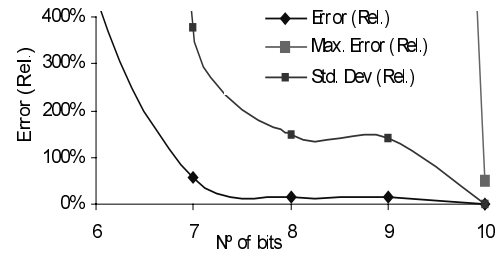


Figure 10 – N° of levels vs Accuracy for Irregular Attribute

The previous discussion concerned the reconstruction error. Another interesting property of quantized attributes is that typical OLAP range-sum queries are answered with negligible error (typically much smaller than the errors associated with the reconstruction of individual values). Figure 11 illustrates this by showing the estimation error of range-sum aggregation queries on a set of quantized attributes. The results are organized by aggregation size (average number of values within the aggregated groups). The errors are extremely small, even for small aggregation sizes, and decrease as the size of the aggregation increases.

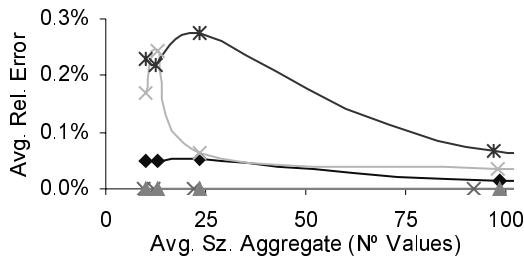


Figure 11 – Size of Aggregation (Nº of values) vs Accuracy for Set of Quantized Attributes

7. Conclusions

In this paper we have proposed the FCompress fact compression strategy that can be used with fact and summary tables or data cubes. It was designed to be integrated in the data warehouse schema with minor changes. Value attributes are replaced by very accurate approximations determined using adaptable quantization guided by user-given error parameters. The technique has been evaluated, showing that very small errors can be achieved for point reconstruction (typically below 2%) and even smaller errors for OLAP queries, while compressing the data sets two to five times. A sensible improvement in performance was also observed.

12. References

- [1] D. Barbara and M. Sullivan, "Quasi-Cubes: A space-efficient way to support approximate multidimensional databases", Technical Report, ISE Dept., September 1997.
- [2] D. Barbara and M. Sullivan, "Quasi-Cubes: Exploiting approximations in multidimensional databases", in Sigmod Record Volume 26, Number 1, March 1997.
- [3] W. G. Cochran. Sampling Techniques. Willey&Sons, New York, 3rd edition, 1977.
- [4] Data Reduction Techniques: Special Issue of Bulletin of the Technical Committee on Data Engineering of the IEEE Computer Society, December 1997, Vol. 20, n 4.
- [5] Furtado, Pedro "Accurate Reduced Representations of Multidimensional Fact-Like Data Sets", PhD thesis, Universidade de Coimbra, 2000.
- [6] J-L Gaily and M. Adler. Zlib home page. <http://quest.jpl.nasa.gov/zlib/>.
- [7] P. B. Gibbons and Y. Matias, "New Sampling-based Summary Statistics for Improving Approximate Query

Answers", in Procs. of ACM SIGMOD Intern. Conference on Management of Data, June 1998, pp 331-342.

[8] J. Goldstein, R. Ramakrishnan, U. Shaft, "Compressing Relations and Indexes", in Procs. of the 14th Int. Conf. on Data Engineering, February 1998, Florida, USA.

[9] J. M. Hellerstein, P. J. Haas, H. J. Wang, "Online Aggregation", in Procs. of ACM-SIGMOD Conf, 1997.

[10] J. Hennessy and D. Patterson, "Computer Architecture: A Quantitative Approach". Morgan Kaufmann, 1990: ISBN 1-55860-069-8.

[11] S.P. Lloyd. "Least Squares Quantization in PCM". IEEE Transactions on Information Theory, IT-28:127-135, March, 1982.

[12] M. Nelson, J-L Gaily, "The Data Compression Book", 2nd ed. 1996 - M&T Books, ISBN 1-55851-434-1.

[13] W. G. Ng, C. V. Ravishankar, "Relational Database Compression Using Augmented Vector Quantization", in Procs. Of the 11th Int. Conf. on Data Engineering, 1995.

[14] V. Poosala, Y. Ioannidis, "Selectivity Estimation Without the Attribute Value Independence Assumption", Procs. of the 23rd VLDB Conf, Athens, Greece, 1997.

[15] M. A. Roth and S. J. Van Horn, " Database Compression" in Sigmod Record, Vol. 22, N° 3, 1993.

[16] K. Sayood, "Introduction to Data Compression", Morgan Kauffman, 1996.

[17] J. S. Vitter, Min Wang, "Data Cube Approximation and Histograms via Wavelets", 7th International Conference on Information and Knowledge Management, Bethesda, Maryland, November 1998.

[18] J. S. Vitter, M. Wang, "Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets", in Proc. of the 1999 Int. Conf. on the Management of Data, Philadelphia, PA, USA, 1999.

[19] Welsh, Terry, "A Technique for High-Performance Data Compression" IEEE Computer, Volume 17, N° 6, June 1984, pages 8-19.

[20] J. Ziv, Lempel, "A Universal algorithm for sequential data compression", IEEE Transactions on Information Theory, Volume 23, N° 3, May 1977, pages 337-343.

[21] Oracle8 Administrator's Guide.