COMPRESSING HIGH-DIMENSIONAL DATA SPACES USING NON-DIFFERENTIAL AUGMENTED VECTOR QUANTIZATION

O.O. Olugbara[1] and A.A. Atayero[2]

Department of Computer and Information Sciences[1]
Electrical & Information Engineering[2]
Covenant University, Ota, Nigeria
oluolugbara@gmail.com, atayero@ieee.org

Abstract

Most data-intensive applications are confronted with the problems of I/O bottleneck, poor query processing times and space requirements. Database compression has been discovered to alleviate the I/O bottleneck, reduce disk space, improve disk access speed, speed up query, reduce overall retrieval time and increase the effective I/O bandwidth. However, random access to individual tuples in a compressed database is very difficult to achieve with most available compression techniques.

We propose a lossless compression technique called non-differential augmented vector quantization, a close variant of the novel augmented vector quantization. The technique is applicable to a collection of tuples and especially effective for tuples with many low to medium cardinality fields. In addition, the technique supports standard database operations, permits very fast random access and atomic decompression of tuples in large collections. The technique maps a database relation into a static bitmap index cached access structure. Consequently, we were able to achieve substantial savings in space by storing each database tuple as a bit value in the computer memory.

Important distinguishing characteristics of our technique is that individual tuples can be compressed and decompressed, rather than a full page or entire relation at a time, (b) the information needed for tuple compression and decompression can reside in the memory or at worst in a single page. Promising application domains include decision support systems, statistical databases and life databases with low cardinality fields and possibly no text fields.

Keywords: Data Compression, High-Dimensional Data Space, Vector Quantization, Database

1. Introduction

Compression has traditionally not been widely used in commercial database systems because many compression methods are effective only on large chunks of data and are thus incompatible with random access to small parts of the data [1]. Many of the available schemes are only suitable for data compression, which differs from database compression because it is usually performed at the granularity of the entire data objects. In data compression, access to random portions of the compressed data is impossible without decompressing the entire file. Evidently, this is not practical for database systems whose essential function is query processing. Efficient query processing and random accessing to small parts of data without incurring serious overhead is only achievable by fine-grained units like tuple or attributes level decompressions. Compression methods that provide fast decompression and random access are therefore, more attractive for databases than schemes that offer better compression effectiveness. The drawback typically associated with compression is that it puts extra burden on the CPU. However, recent works on database compression [1, 2] have shown that it:

improves system performance especially in read-intensive environments,

provides significant improvement in query processing performance,

reduces disk seek times,

increases disk bandwidth,

reduces network communication costs in distributed applications,

increases buffer hit rate and

decreases disk I/O to log devices.

The problem on which this work premises is in Augmented Vector Quantization (AVQ) [3], which was also called Attribute Enumerative Coding (AEC) [4] or Tuple Differential Coding (TDC) [5, 6]. The goal of the study is to adapt AVQ to address the problem of randomly accessing and individually decompressing tuples, while maintaining compact storage of the data [7]. The original AVQ like many block-oriented schemes such as Adaptive Text Substitution (ATS) [8] compresses and decompresses relation tuples that are locally confined to memory blocks. The problems here are:

block (or page) level compression can result in poor query processing times,

compressed block can cross disk block boundaries and

the size of a compressed block can change when data in a block is updated [2].

Furthermore, random access to individual tuple is still not possible until a block of memory is decompressed. In addition, tuple ordering and differencing in AVQ present overhead cost that are problematic for designing lightweight compression and decompression routines especially when the database is unstable. Both tuple and attributes level compressions were shown to be more attractive from the query processing view point [2]. However, attributes level compression performs better, but has poor compression ratio. The query processing power of tuple level compression, which gives higher compression rate, can be improved upon. This is the motivation for the present work. The solution we propose for randomly accessing stored tuples in a compressed relation is a static bitmap index structure.

The rest of the paper is briefly organized as follows. Section 2 provides the necessary background information on traditional AVQ. Section 3 describes the new scheme and an analogue of bitmap index is suggested for its implementation rather than using expensive B-tree index structure. The evaluation of the method is considered in Section 4 and the paper is concluded in Section 5 with a brief note.

2. AVQ Overview

The AVQ represents a series of tuple values in a relational database $R = < A_1, A_2, \dots, A_n >$ by the differences between them, where A_i, i=1(1)n are n sets of natural numbers. The method is particularly applicable to sets and databases and it works as follows. First, each tuple in R is treated as integer and R is then sorted by rows. Successive tuples are then differenced and the differences are used to represent R. This technique is formally defined by the quantizer Q_L as follows [3]:

Given a vector quantizer:

$$Q : R \to Z^+, \; Q_L : R \to Z^+ \times N_R$$

is a lossless mapping that encodes a tuple $t \in R$ by the pair <C(t), d(t, Q(t))>, where C is the coder that produces the codeword denoting Q(t) and the difference d between any two tuples is given by Equation (1).

$$d(t_i, t_j) = \begin{cases} \varphi(t_j) - \varphi(t_i), & \text{if } t_i < t_j \\ \varphi(t_i) - \varphi(t_j), & \text{otherwise} \end{cases} \tag{1}$$

The compression efficiency of the technique depends on the choice of the codebook. If the codebook is properly designed, the average difference between a tuple and its representative tuple will be small enough that it takes fewer bits to encode than the original tuple. The simplest form of AVQ algorithm is described according to the following steps [3]:

Step 1: Attribute encoding

This is the first preprocessing stage and it achieves compression by mapping a long string of characters attribute to a short number.

Step 2: Attribute domain ranking

The lexicographical order defined by function φ is dependent on the ordering of the attribute domains. Different domain orderings give rise to different orders and different orderings of tuples also give rise to different amount of differences among tuples ordinals, thus affecting the amount of compression.

Step 3: Tuple reordering

Every tuple in R is totally ordered via an ordering rule. The rule is usually a lexicographical order with respect to the attribute sequence in R defined by the function $\varphi : R \to N_R$, where $N_R = \{0, 1, \dots, \|R\| - 1\}$ and the number $\| R \| = \prod_{i=1}^{n} | A_i |$ is the size of the R space. The function φ is defined for every tuple $t = < a_1, a_2, \dots, a_n > \in R$ by:

$$\varphi(t) = \sum_{i=1}^{n} \left(a_i \prod_{j=i+1}^{n} | A_j | \right) \tag{2}$$

The inverse function φ^{-1} defined for all $e \in N_R$ and i = 1(1)n-1 is given by:

$$\varphi^{-1}(e) = < t > \tag{3}$$

Where

$$a_0^r = e, \quad a_n = a_{n-1}^r,$$

$$a_i = \left\lfloor \frac{a_{i-1}^r}{\prod_{j=i+1}^{n} | A_j |} \right\rfloor \tag{4}$$

$$a_i^r = a_{i-1}^r - a_i \prod_{j=i+1}^{n} | A_j | \tag{5}$$

The function φ converts each t ∈ R to a unique integer φ(t) that represents its ordinal position within the R space and a total order is based on this function. To avoid the use of auxiliary variables a_i^r for i = 1(1)n, we replace Equations (4) and (5) by an alternative model (Equation 6) that randomly computes a given a_i from e and previously computed a_{i-j}, j = 1(1)i-1 for all i = 1(1)n. The symbols $\lfloor x \rfloor$ and $\lceil x \rceil$ denote the usual floor and ceiling functions respectively.

$$a_i = \left\lfloor \frac{e - \sum_{j=1}^{i-1} (a_j \prod_{k=j+1}^{n} | A_k |)}{\prod_{k=i+1}^{n} | A_k |} \right\rfloor \tag{6}$$

Step 4: Block partitioning

The ordered relation is partitioned into disjoint blocks of tuples and the size of a memory page is chosen as the partition size. When a tuple is required, the block it resides in is transferred from the disk to the main memory. Coding and decoding of tuples are localized to block level granularity.

Step 5: Block encoding

A particular block consists of a set of ordered tuples and a representative tuple $\bar{t}_k$ is chosen from the block so as to minimize total distortion. Each tuple is therefore, replaced by its difference from $\bar{t}_k$ to obtain numerically smaller tuples with fewer bytes of storage. The leading zero components in each difference are encoded using run-length coding.

3. Non-differential AVQ

The new compression algorithm called Non-differential AVQ (NAVQ) is directly based on AVQ and it compresses a given relation at tuple level granularity. Compression is done while tuples are being stored, thus it supports randomize decompression of individual tuples. The method uses modular arithmetic [9] to associate a tuple t with a pair of integers (q, r), where $0 \leq q < m$ and $0 \leq r < n$. The modular arithmetic partitions the set $\aleph$ of natural numbers into n equivalent classes (or strata), where $n \geq 2 \in \aleph$. Each stratum has $m \geq 1 \in \aleph$ tuples with common features and the entire relation is mapped into a static bitmap index structure $B_{m,n}$. The pair (q, r) serves as the position marker in $B_{m,n}$ for t. We use the term static to discriminate the array from the conventional bitmap index [10], here called dynamic bitmap. The difference between the two bitmap types is that static bitmap is statically created from predefined statistics of the relation and dynamic bitmap is dynamically created from the elements of the relation.

NAVQ is formally defined by a database quantizer Q_N as follows. Given a vector quantizer $Q : R \rightarrow \aleph,\ Q_N : \aleph \rightarrow \aleph \times \aleph$ is a lossless contraction mapping that encodes each $t \in R$ by the pair $< C_1(t), C_2(t) >$<C1(t), where C_1 and C_2 are coders that produce pair of codewords denoting the position of Q(t) in a bitmap. Apparently, $Q_N(Q(t))$ is a mapping composition and the most important distinguishing aspects of the technique are (a) each compressed tuple can randomly be decompressed at a time complexity independent on tuples size, (b) compression and decompression can be carried out without referencing the entire page, let alone the entire relation.

NAVQ is a lossless vector quantization that maps tuple values to bit values. It is different from AVQ since it does not represent a series of tuple values by their differences and does not perform tuple sorting, which can lead to a performance overhead for unstable database. Both methods are similar because they are applied by first treating each tuple in a database table as an integer. Both algorithms use the same mapping φ to convert a tuple to a unique codeword. The compression algorithm is clearly described as follows.

```
Algorithm 1: DB_Compressor
  Input:
      The relation R to compress
  Output:
      A static bitmap structure B_{m,n} of compressed tuples, where $m = \lceil \| R \| / n \rceil$ and $n \in (1, \|R\|)$ are fixed constants.
  Method:
      This algorithm is basically in phases of preprocessing as follows.
    If R is compressible Then
    1. Create B_{m,n} and initialize its entries to 0
    2. If attribute encoding is required Then perform attribute encoding on tuples in R
    3. Map every $t \in R$ to a codeword $e = \varphi(t)$ using Equation 2
    4. Map e to B_{m,n} using $B_{q,r} = 1$, where $q = \lfloor e/n \rfloor$ and $r = MOD(e/n)$
```

End_Algorithm DB_Compressor

The decompression technique is based on the inversion function given by Equation (3) and it works directly in opposite mode to the compression routine. The detail description of the algorithm is given below.

```
Algorithm 2: DB_Decompressor
 Input:
     $B_{m,n}$ and n.
 Output:
     A list of $k \geq 1$ tuples $t_1, t_2, \ldots, t_k$.
 Method:
     This simple algorithm is the direct inverse of DB_Compressor
   If R was compressed Then
   1. Create the actual relation R
   2. Convert each bitmap entry (q, r) with value 1 to a codeword e using $e = q * n + r$
   3. Perform codeword decoding on e using Equation 3 to obtain $t \in R$
   4. If attribute encoding was carried out Then perform attribute decoding on t
   5. Insert t into R
  End_Algorithm DB_Decompressor
```

3.1. Application of NAVQ Algorithm

The algorithm was applied to compress the relation R given in [5]. The elements of N_R and the corresponding entries of the bitmap are displayed in Table I. ||R|| = 262144, n = 5 and the efficiency of the technique is 94.44%. Any value of n can be chosen, but for the bitmap to attain high degree of storage utilization, small values are appropriate. We recommend the cardinality of the relation as an appropriate value of n.

Table I: A Relation R

N_R	(q, r)	N_R	(q, r)	N_R	(q, r)	N_R	(q, r)
14816	(2963, 1)	92696	(18539, 1)	154073	(30814, 3)	212130	(42426, 0)
18984	(3796, 4)	100950	(20190, 0)	158233	(31646, 3)	216867	(43373, 2)
21140	(4228, 0)	105118	(21023, 3)	162206	(32441, 1)	223316	(44663, 1)
39331	(7866, 1)	110105	(22021, 0)	173803	(34760, 3)	227484	(45496, 4)
43117	(8623, 2)	117795	(23559, 0)	179038	(35807, 3)	232022	(46404, 2)
47252	(9450, 2)	125352	(25070, 2)	182804	(36560, 4)	235363	(47072, 3)
51104	(10220, 4)	128798	(25759, 3)	186841	(37368, 1)	244658	(48931, 3)
68702	(13740, 2)	134302	(26860, 2)	190996	(38199, 1)	248414	(49682, 4)
80419	(16083, 4)	137827	(27565, 2)	204052	(40810, 2)	252190	(50438, 0)
85140	(17028, 0)	149920	(29984, 0)	207828	(41565, 3)	255449	(51089, 4)

3.2. Data Structure and Operations

We now consider how access mechanisms are constructed on coded tuples and how the tuples can be retrieved and modified. The focus is to give an idea of how the method can be integrated with standard access and retrieval mechanisms. We restrict attention to basic operations rather than to general queries because all queries, simple or complex, reduce to a set of basic tuple operations [5].

We propose a static bitmap index as a suitable data structure for efficient implementation of the algorithms. Dynamic bitmap index is a special kind of index structure consisting in arrays of bits. Each bitmap represents one of the values in the indexed column and the bit position in the array corresponds to the row position in the table [11]. Bitmap indexes are most desirable for low cardinality field relations, systems with low concurrency, few updates and searches. They are frequently used in data warehouses since all of these conditions are found there [11]. A bitmap structure occupies much less space than a correspondent B-tree index [12, 13], an alternative structure that can be used instead of the bitmap. The primary key will then be the q values, which can be made small by choosing n to be large. However, the overhead incur by the q values and pointers to the next node will jeopardize the essence of compression and so bitmap is the most suitable structure for the application since these overheads are avoided.

Static bitmap supports high concurrency, many updates and frequent searches. The structure consisting in arrays of bits, but each array represents a stratum of tuples with common features and not values in the index column. A static bitmap is different from other cached (or pre-computed) access structures in the sense that it does not hold information about relations, but a one–to–one correspondence exists between it and the original relation. The great advantage of static bitmap index is that it allows the database system to avoid direct reading from or writing to the relation. The information needed to answer the query is not taken from the relation and thus in this context, it is related to materialized views [14].

We illustrate the two structures by a concrete example using the Sell relation [15] shown in Table II. The dynamic bitmap indexes and static bitmap index for this relation are respectively given by Tables III and IV.

Table II: Sell Relation

Company	**Product**	**Country**
IBM	PC	France
IBM	PC	Italy
IBM	PC	UK
IBM	Mainframe	France
IBM	Mainframe	Italy
IBM	Mainframe	UK
DEC	PC	France
DEC	PC	Spain

DEC	PC	Ireland
DEC	Mini	France
DEC	Mini	Spain
DEC	Mini	Ireland
ICL	Mainframe	Italy
ICL	Mainframe	France
…	…	…

Table III: Dynamic Bitmap Indexes

(a) Index Company			**(b) Index Product**			**(c) Index Country**				
IBM	DEC	ICL	PC	MA	MI	FR	IT	UK	SP	IR
1	0	0	1	0	0	1	0	0	0	0
1	0	0	1	0	0	0	1	0	0	0
1	0	0	1	0	0	0	0	1	0	0
1	0	0	0	1	0	1	0	0	0	0
1	0	0	0	1	0	0	1	0	0	0
1	0	0	0	1	0	0	0	1	0	0
0	1	0	1	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	0	1	0
0	1	0	1	0	0	0	0	0	0	1
0	1	0	0	0	1	1	0	0	0	0
0	1	0	0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0	0	0	1
0	0	1	0	1	0	0	1	0	0	0
0	0	1	0	1	0	1	0	0	0	0

Table IV: Static Bitmap Index

	r_0	r_1	r_2
q_0	1	1	1
q_1	0	0	1
q_2	1	1	0
q_3	0	0	0
q_4	0	0	0
q_5	1	0	0
q_6	1	1	0
q_7	0	0	0
q_8	0	1	0
q_9	0	1	1
q_{10}	0	0	0
q_{11}	0	0	1
q_{12}	1	0	0
q_{13}	0	0	0
q_{14}	0	0	0

3.3. Access method

A static bitmap index structure is constructed using the data in Table (I). Suppose a query wishes to locate the tuple <2, 1, 3, 38, 30>, the query only needs to check if the location given by (32441, 1) for this tuple in the bitmap has a bit value one. The value of one is an indication that the tuple is present in the relation while zero means it does not exist.

3.4. Tuple insertion, deletion and modification

Tuple insertions and deletions are supported in the compressed database as follows. Suppose we wish to insert the tuple <2, 3, 1, 39, 24>. The codeword for the tuple is 186840 and the tuple is stored as 1 in location (37386, 0) in the bitmap structure. For tuple deletion we simply assign bit value 0 to this location. Tuple modification is simply a combination of tuple insertion and deletion. Conclusively, standard database operations remain the same even when the database is NAVQ coded.

4. Evaluation of the Techniques

The performance of a compression technique can be measured in terms of (a) compression ratio, (b) compression and decompression time overhead and (c) query response time. We concentrate on the first factor in this paper since our access structure provides efficient access mechanism.

The efficiency μ of a compression technique operating on a relation R with k tuples is usually defined in terms of two parameters D and C respectively denoting the sizes of the relation before and after compression. The ratio μ is defined by [5]:

$$\mu = 1 - \frac{C}{D} \tag{7}$$

Equation (7) suggests that positive efficiency is not always guaranteed as μ may take on negative values depending on whether the relation is compressible or not. If C > D, negative efficiency occurs and the database is said to be incompressible. A positive efficiency implies the technique compresses well according to the largeness of μ. Small positive value of μ signifies poor compression and zero value shows that compression is not achieved by the technique.

The efficiencies of two compression techniques can also be compared using Equation (7). In this case C and D are respectively the sizes of the compressed relation when techniques 1 and 2 are applied. The value of μ being zero implies that both techniques have the same efficiencies, small value of μ is an indication that the second technique performs better than the first and large value of μ means the first technique compresses better than the second. We further discriminate between two techniques with nearly the same compression efficiency using other characteristics such as compression throughput, simplicity and efficiency of the algorithms. In most cases, a lightweight scheme is preferable to a heavyweight scheme.

4.1 Compression efficiency of AVQ

The efficiency of AVQ is affected by two factors compression overhead per tuple and tuple spacing [5]. The compression overhead per tuple is the size of the count field used to indicate the number of leading zero components of a tuple. Usually, a fixed-size field of size α bits is used to encode this count in order to avoid making the scheme overly complex. For n attribute domains, the number of leading zero components in any difference tuple is larger than zero, but less than n. Thus, the number of bits required for the field is $\alpha = \lceil \log_2 n \rceil$. If relation R has k tuples, the total compression overhead is given by α(k-1), since k tuples yield k-1 differences.

The spacing between two tuples $t_i < t_j$ with respect to φ is measured by a function

$\delta : R \times R \to [0,1)$ defined as:

$$\delta(t_i,t_j)=\lceil \log_2(\varphi(t_j)-\varphi(t_i))\rceil \tag{8}$$

The quantity $\delta(t_i,t_j)$ measures the number of bits required to represent the numerical difference between tuples t_i and t_j. The further apart the tuples are, the larger is the difference. The total space requirement in bits for the k-1 tuple differences and fixed compression overhead per tuple is:

$$C=\sum_{i=1}^{k-1}(\delta(t_{i-1},t_i)+\alpha)$$

The size of the relation before compression is:

$$D=k\lceil \log_2\varphi(t)\rceil$$

The compression efficiency ratio of AVQ is given by:

$$\mu=1-\frac{\alpha(k-1)+\sum_{i=1}^{k-1}\delta(t_{i-1},t_i)}{k\lceil \log_2\varphi(t)\rceil} \tag{9}$$

4.2. Compression efficiency of NAVQ

The efficiency of NAVQ can be determined ahead of compression and it is affected by two factors namely the norm of R space and the overhead of storing the parameter n, which corresponds to column size of the bitmap. Since every tuple is mapped to a bit value, tuple size is just one bit. This follows that ||R|| bits are required for storing a database relation with a maximum of ||R|| tuples. If n is chosen to correspond to the cardinality of R, then the efficiency μ* of the method assuming a low size dictionary was used for attribute encoding, is given by:

$$\mu^*=1-\frac{\|R\|+\alpha}{k\lceil \log_2\varphi(t)\rceil} \qquad 10)$$

It can easily be shown that NAVQ gives higher compression ratio than AVQ whenever the load factor of the relation (i.e. the ratio of number of records to the size of the relation) is high. The question of which of the two schemes gives a higher compression rate can be solved as follows. If for a given k, the load factor β satisfies condition (11) then NAVQ gives a higher compression rate than AVQ, otherwise the reverse is the truth. Even if the compression rate of AVQ is higher (for low tuples in relation), NAVQ has the advantage of providing random access to individual tuples in the relation.

$$\beta>\frac{k}{\alpha(k-2)+\sum_{i=1}^{k-1}\delta(t_{i-1},t_i)} \tag{11}$$

5. Conclusion

This paper presented a new compression algorithm that is based on AVQ and demonstrates its effectiveness on relational database, which exhibits low to medium cardinality fields and numeric fields. The algorithm supports standard database operations, permits very fast random access and atomic decompression of tuples in large collection of data with low decompression cost.

In comparison to a novel AVQ, our technique hopefully yields a higher compression ratio for large tuples. However, in general, the technique has the disadvantage that it compresses only low cardinality field database relations. We hope to develop a hybrid version of this algorithm to compress databases containing generic-purpose data, such as images, sound and text. We also intend to extend the technique for mining association rules in compressed databases.

References

[1] Z. Chen, J. Gehrke and F. Korn, "Query Optimization in Compressed Database Systems", ACM SIGMOD, USA, 2001, pp. 271-282.

[2] G. Ray, J.R. Haritsa and S. Seshadri, "Database Compression: A Performance Enhancement Tool", COMAD, 1995.

[3] W.K. Ng and C.V. Ravishankar, "Relational Database Compression Using Augmented Vector Quantization", Proceedings of the 11th IEEE International Conference on Data Engineering, 1995, pp. 540-549.

[4] W.K. Ng and C.V. Ravishankar, "Attribute Enumerative Coding: A Compression Technique for Tuple Data Structures", Proceedings of the 4th Data Compression Conference, 1994, pp. 461.

[5] W.K. Ng and C.V. Ravishankar, "Block-oriented Compression Techniques for Large Statistical Databases", IEEE Trans. Knowledge and Data Eng, 1997, pp. 9, 314-328.

[6] W.B. Wu and C.V. Ravishankar, "The Performance of Difference Coding for Sets and Relational Tables", Journal of the ACM, Vol. 50, 2003, pp. 665-693.

[7] A. Cannane and H.E. Williams, "A Compression Scheme for Large Databases", ACSC. vol. 10, 2000, pp. 241-248.

[8] T.A. Welch, "A Technique for High Performance Data Compression", Computer, vol. 7, No. 6,1984, pp. 8-19.

[9] R.P. Grimaldi, Discrete and Combinatorial Mathematics, An Applied Introduction 2nd ed., Addisson-Wesley, USA, 1989.

[10] C.Y. Chan and Y.E. Ioannidis, "Bitmap Index Design and Evaluation", Proc. SIGMOD Conference, 1998, pp.355-366.

[11] P. Bizarro and H. Madeira, "The Dimension-Join: A New Index for Data Warehouses", SBBD 2001, pp. 259-273.

[12] R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indexes", Acta Informatica1(3), 1979, pp. 173-189.

[13] D. Comer, "The Ubiquitous B-tree", Computing Surveys 2(2), 1979, pp. 122.

[14] N. Roussopoulos, "Materialized Views and Data Warehouses", ACM SIGMOD Record 27(1), 1998, pp. 21-26.

[15] J.G. Hughes, Database Technology, A Software Engineering Approach. Prentice-Hall, 1988.