Relational Database Compression Using Augmented Vector Quantization*†

WEE K. NG   CHINYA V. RAVISHANKAR

Department of Electrical Engineering and Computer Science
The University of Michigan, Ann Arbor, MI 48109-2122
E-mail: {wkn,ravi}@eecs.umich.edu

Abstract

Data compression is one way to alleviate the I/O bottleneck problem faced by I/O-intensive applications such as databases. However, this approach is not widely used because of the lack of suitable database compression techniques. In this paper, we design and implement a novel database compression technique based on vector quantization *(VQ). VQ is a data compression technique with wide applicability in speech and image coding [3, 5], but it is not directly suitable for databases because it is lossy. We show how one may use a* lossless *version of vector quantization to reduce database space storage requirements and improve disk I/O bandwidth.*

1 Introduction

Processor speed, memory speed, and memory size have grown exponentially over the past few years. However, disk speeds have improved at a far slower rate. As a result, many applications such as database systems are now limited by the speed of their disks rather than the power of their CPUs [1]. As improvements in processor and memory speeds continue to outpace improvements in disk speeds, the severity of the I/O limitation will only increase.

One way to alleviate this problem is through database compression [2, 10], which will not only reduce the space requirements, but also increase the effective I/O bandwidth since more data is transferred in compressed form. However, this approach has not been widely adopted. One reason is the lack of suitable *database* compression techniques. Database compression differs from data compression in general. Conventional data compression is usually performed at the granularity of entire data objects. Access to random portions of the compressed data objects is impossible without decompressing the entire object. Clearly, this is not practical for database systems. What is required is a technique that not only compresses data well, but also supports standard database operations.

In this paper, we design a novel database compression technique that has the following characteristics: (1) it is lossless, (2) it compresses/decompresses locally, thus permitting localized access to compressed data, and (3) it supports standard database operations as described above. The technique is based on the concept of *Vector Quantization* (VQ). Conventional VQ is a *lossy* data compression technique with wide applicability in speech and image coding [3, 5]. We propose a *lossless* version called Augmented Vector Quantization (AVQ) that is appropriate for database compression. We have also restricted our attention to relational databases as they are very widely used.

This paper is organized as follows. Section 2 provides background material for conventional lossy vector quantization. We also discuss how VQ may be adapted for database compression. In Section 3, we describe issues in the practical implementation of AVQ. Section 4 illustrates how AVQ supports standard database operations. We evaluate the performance of AVQ in Section 5. Finally, the last section concludes the paper.

2 Augmented Vector Quantization

2.1 Conventional vector quantization

Multidimensional vector quantization (or VQ) is a technique for lossy encoding of n-dimensional vectors.

*This work was supported in part by the Consortium for International Earth Science Information Networking.

†The material contained in this paper may be covered by a pending patent application [11].

1063-6382/95 $4.00 © 1995 IEEE

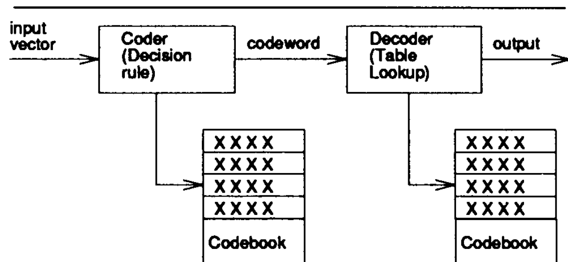

Figure 2.1: Block diagram of a vector quantizer.

Conventional VQ operates as follows. Let X be an (possibly infinite) input set of n-dimensional vectors. First, a fixed and finite set Y of n-dimensional output vectors is selected, with $|X| > |Y|$. Each $\boldsymbol{x}_i \in X$ is mapped into a $\boldsymbol{y}_j \in Y$ which approximates it according to some suitable criterion, and $\boldsymbol{y}_j$ is output in place of $\boldsymbol{x}_i$. Since the set Y of output vectors is finite, a *codeword* j can be used to identify each output vector $\boldsymbol{y}_j$ representing an original input vector $\boldsymbol{x}_i$. Compression is achieved because the size of each codeword j is smaller than that of the corresponding $\boldsymbol{x}_i$. During decoding, the codeword j is simply replaced by the output vector $\boldsymbol{y}_j$, approximately reconstructing the original input vector $\boldsymbol{x}_i$. The coding/decoding process is illustrated in Figure 2.1. A brief formalization is given below:

A vector quantizer $Q : \mathbb{R}^n \rightarrow Y$ is a mapping of n-dimensional Euclidean space $\mathbb{R}^n$ into a finite subset Y of $\mathbb{R}^n$, where $n > 1$, $Y = \{\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_m\}$. Each $\boldsymbol{y}_i \in \mathbb{R}^n$ is an output vector.

Y induces a partition of $\mathbb{R}^n$: $R_1, R_2, \ldots, R_m$ where $R_i = Q^{-1}(\boldsymbol{y}_i) = \{\boldsymbol{x} \in \mathbb{R}^n \mid Q(\boldsymbol{x}) = \boldsymbol{y}_i\}$. Thus, $\bigcup_{i=1}^m R_i = \mathbb{R}^n$ and $R_i \cap R_j = \phi$ for all $i \neq j$. The quantizer is uniquely defined by the output set Y and the corresponding partition $\{R_i\}$.

The quantizer can also be seen as a combination of two functions: a coder and a decoder. The coder C is a mapping of $\mathbb{R}^n$ into the index or codeword set $J = \{1, 2, \ldots, m\}$ and the decoder D is a mapping of J into the output set Y.

A distortion measure $d(\boldsymbol{x}, \hat{\boldsymbol{x}})$ represents the penalty associated with reproducing vectors $\boldsymbol{x}$ by $\hat{\boldsymbol{x}} = Q(\boldsymbol{x})$. A commonly used measure is the *square error* between $\boldsymbol{x}$ and $\hat{\boldsymbol{x}}$ defined as follows:

$$d(\boldsymbol{x}, \hat{\boldsymbol{x}}) = \sum_{i=1}^{n} (\boldsymbol{x}_i - \hat{\boldsymbol{x}}_i)^2 \qquad (2.1)$$

The optimal quantizer Q is the one that minimizes $d(\boldsymbol{x}, \hat{\boldsymbol{x}})$ for all input vectors $\boldsymbol{x}$ [3].

The set Y of output vectors is also called the *codebook*. Given a set of input vectors, the design of optimal codebooks has been extensively studied by Linde, Buzo and Gray [9]. They have proposed an algorithm that determines the optimal codebook via *iterative refinements*. This method is inefficient as it requires a non-deterministic number of iterations. As we shall see, our adaptation of VQ (AVQ) has a definite advantage over their algorithm: It computes the codebook in constant time.

Another issue that determines the performance of VQ is the structure of the codebook. During decoding, the codebook should be structured as to permit efficient searching of codewords. For large codebooks, this search process can become computationally intensive. Many structures have been proposed to reduce the search time [5]. In this respect, AVQ has another advantage: *No* searching is required.

2.2 Database compression

A relational database is a natural candidate for the application of VQ. A relation is a table of n-tuples, each of which is a vector, or a point in n-dimensional space. We will use the terms *tuple* and *vector* interchangeably. A direct application of VQ to encode a relation would be to find a set of *representative* tuples, and replace each tuple in the relation with a codeword or index that indicates the representative tuple that is *closest* to it. Unfortunately, this method of coding is lossy; the original tuples are no longer completely recoverable. Thus, a new design is needed.

We propose Augmented Vector Quantization (AVQ), a lossless database compression technique based on VQ. Instead of replacing each tuple in a relation only by its codeword as VQ does, we also include the *difference* between the tuple and its representative tuple. The method is formally defined below. But first, we need some preliminaries in relational database terminology.

A relation scheme $\mathcal{R} = \langle\langle A_1, A_2, \ldots, A_n \rangle\rangle$ is the cartesian product of the set of attributes A_i, i.e., $\mathcal{R} = A_1 \times A_2 \times \cdots \times A_n$. It corresponds to the n-dimensional space $\mathbb{R}^n$ as defined in the previous section except that the size of each dimension $|A_i|$, may not be the same, and the value $a \in A_i$ within each dimension or domain is non-negative. A relation R is a subset of $\mathcal{R}$ and corresponds to the set of input vectors to be coded. A tuple $t \in R$ is an n-dimensional vector.

All points in $\mathcal{R}$ may be totally ordered via an ordering rule. Let $\mathcal{N}_{\mathcal{R}} = \{0, 1, \ldots, ||\mathcal{R}|| - 1\}$ be a set of integers that correspond to $\mathcal{R}$, where $||\mathcal{R}|| = \prod_{i=1}^n |A_i|$ is

the size of the $\mathcal{R}$ space. Define a function $\varphi : \mathcal{R} \rightarrow \mathcal{N}_{\mathcal{R}}$ as follows:

$$\varphi(a_1, a_2, \ldots, a_n) = \sum_{i=1}^{n} \left(a_i \prod_{j=i+1}^{n} |A_j| \right) \quad (2.2)$$

for all $\langle a_1, a_2, \ldots, a_n \rangle$[1] $\in \mathcal{R}$. The inverse of φ is defined as:

$$\varphi^{-1}(e) = \langle a'_1, a'_2, \ldots, a'_n \rangle \quad (2.3)$$

for all $e \in \mathcal{N}_{\mathcal{R}}$ and $i = 1, 2, \ldots, n-1$,

$$a'_i = \left\lfloor \frac{a^r_{i-1}}{\prod_{j=i+1}^{n} |A_j|} \right\rfloor \quad (2.4)$$

$$a^r_i = a^r_{i-1} - a'_i \prod_{j=i+1}^{n} |A_j| \quad (2.5)$$

where $a^r_0 = e$ and $a'_n = a^r_{n-1}$.

φ is a n-dimensional to 1-dimensional mapping that maps a tuple $t \in \mathcal{R}$ uniquely into its *ordinal* position in the $\mathcal{R}$ space. Given two tuples $t_i, t_j \in R$, we may define a total order based on φ, denoted by $t_i \prec t_j$, such that t_i precedes t_j if and only if $\varphi(t_i) < \varphi(t_j)$.

With these preliminaries, the difference between any two tuple t_i, t_j may be defined as:

$$d(t_i, t_j) = \begin{cases} \varphi(t_j) - \varphi(t_i) & \text{if } t_i \prec t_j \\ \varphi(t_i) - \varphi(t_j) & \text{otherwise} \end{cases} \quad (2.6)$$

AVQ is defined by the quantizer Q_L as follows:

Definition 2.1 (AVQ) . *Given a vector quantizer $Q : \mathcal{R} \rightarrow \mathbb{Z}^+$, $Q_L : \mathcal{R} \rightarrow \mathbb{Z}^+ \times \mathcal{N}_{\mathcal{R}}$ is a lossless mapping that encodes a tuple $t \in R$ by the pair $\langle C(t), d(t, Q(t)) \rangle$, where C is the coder that produces the codeword (or index into the codebook) denoting $Q(t)$.*

Let $\beta[x]$ denote the minimum number of bits in the binary representation of number x. If $\beta[C(t_i)] + \beta[d(t_i, Q(t_i)] < \beta[t_i]$, then compression is achieved. The compression efficiency of AVQ 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.

We have so far claimed that AVQ is lossless. This is shown in the following theorem:

[1] A tuple is generally enclosed in angle brackets. When used as an argument of a function, the angle brackets are omitted when no confusion arises.

Theorem 2.1 (Lossless property) *AVQ is lossless.*

Proof: During the coding process, a tuple t is quantized into a codeword $C(t)$ and a difference $d(t, Q(t))$. By definition of the difference measure (Equation 2.6),

$$\begin{aligned} d(t, Q(t)) &= \varphi(Q(t)) - \varphi(t) \\ \varphi(t) &= \varphi(Q(t)) - d(t, Q(t)) \end{aligned}$$

assuming $t \prec Q(t)$. During decoding, $C(t)$ indicates that $Q(t)$ is the output vector. By subtracting the difference from $\varphi(Q(t))$, one obtains $\varphi(t)$. From the definition of φ (Equations 2.2 and 2.3), φ is a bijection. Thus, $\varphi(t)$ is uniquely mapped back to the tuple t which is then completely recovered. The same argument holds when $Q(t) \prec t$. ∎

3 Implementation of AVQ

We now see how AVQ may be adapted for database compression. In particular, we are interested in how a relation is encoded and allocated physically to disk blocks. Sections 3.1–3.4 illustrate the steps in the process of transforming a relation into a set of losslessly quantized tuples. Throughout this section and the rest of the paper, we use the relation described in Example 3.1 to illustrate the concepts involved.

Example 3.1 Table (a) in Figure 2.2 shows a relation R with five attribute domains A_1, A_2, A_3, A_4, A_5 denoting the *department*, *job title*, *years in company*, *hours worked per week*, and *employee number* respectively. The size of each domain, i.e., the number of attribute values, is 8, 16, 64, 64, 64 respectively. Table (b) shows the same relation, except that the attribute values have all been encoded to numbers. Attribute encoding is discussed briefly below. The relation in the figure has been partitioned into blocks. Each block is coded/decoded individually. ∎

3.1 Attribute encoding

A variety of attribute domain types are encountered in practice. Attributes such as social security numbers or zip codes are numeric, while those such as names and addresses are alphanumeric, occurring in the form of ASCII characters. The first preprocessing step in AVQ encodes each attribute value to a number. For discrete finite domains where all the attribute values are known in advance, each attribute value is mapped to its *ordinal* position in the domain. For other domain types, more work is needed. For alphanumeric

A_1	A_2	A_3	A_4	A_5
production	*part-time*	24	32	00
marketing	*director*	12	31	01
management	*worker1*	29	21	02
marketing	*worker2*	30	42	03
management	*supervisor*	27	27	04
production	*secretary*	23	25	05
production	*secretary*	34	28	06
production	*worker1*	32	37	07
marketing	*worker2*	39	37	08
production	*executive*	31	25	09
marketing	*part-time*	19	21	10
production	*secretary*	28	22	11
production	*manager*	32	34	12
marketing	*manager*	38	34	13
marketing	*worker2*	26	32	14
personnel	*supervisor*	33	22	15
production	*part-time*	34	28	16
marketing	*part-time*	25	27	17
marketing	*manager*	41	28	18
production	*manager*	32	25	19
marketing	*secretary*	39	29	20
marketing	*manager*	50	26	21
production	*manager*	31	33	22
personnel	*manager*	26	32	23
production	*worker1*	34	26	24
personnel	*worker2*	45	16	25
production	*worker2*	39	37	26
marketing	*worker1*	40	27	27
marketing	*supervisor*	30	44	28
production	*manager*	24	30	29
marketing	*worker2*	33	32	30
marketing	*part-time*	32	42	31
personnel	*supervisor*	19	31	32
production	*part-time*	27	26	33
production	*supervisor*	32	30	34
production	*manager*	36	39	35
management	*worker1*	26	20	36
production	*part-time*	26	27	37
production	*supervisor*	35	25	38
marketing	*supervisor*	39	33	39
production	*worker2*	35	28	40
marketing	*manager*	32	24	41
marketing	*manager*	31	24	42
marketing	*supervisor*	35	19	43
marketing	*executive*	55	23	44
marketing	*manager*	32	27	45
production	*worker2*	37	31	46
personnel	*secretary*	24	26	47
production	*worker2*	30	32	48
marketing	*worker2*	39	31	49

Table (a)

A_1	A_2	A_3	A_4	A_5
3	09	24	32	00
4	12	12	31	01
2	06	29	21	02
4	07	30	42	03
2	10	27	27	04
3	05	23	25	05
3	05	34	28	06
3	06	32	37	07
4	07	39	37	08
3	04	31	25	09
4	09	19	21	10
3	05	28	22	11
3	08	32	34	12
4	08	38	34	13
4	07	26	32	14
5	10	33	22	15
3	09	34	28	16
4	09	25	27	17
4	08	41	28	18
3	08	32	25	19
4	05	39	29	20
4	08	50	26	21
3	08	31	33	22
5	08	26	32	23
3	06	34	26	24
5	07	45	16	25
3	07	39	37	26
4	06	40	27	27
4	10	30	44	28
3	08	24	30	29
4	07	33	32	30
4	09	32	42	31
5	10	19	31	32
3	09	27	26	33
3	10	32	30	34
3	08	36	39	35
2	06	26	20	36
3	09	26	27	37
3	10	35	25	38
4	10	39	33	39
3	07	35	28	40
4	08	32	24	41
4	08	31	24	42
4	10	35	19	43
4	04	55	23	44
4	08	32	27	45
3	07	37	31	46
5	05	24	26	47
3	07	30	32	48
4	07	39	31	49

Table (b)

A_1	A_2	A_3	A_4	A_5	$\mathcal{N}_{\mathcal{R}}$
2	06	26	20	36	10069284
2	06	29	21	02	10081602
2	10	27	27	04	11122372
3	04	31	25	09	13760073
3	05	23	25	05	13989445
3	05	28	22	11	14009739
3	05	34	28	06	14034694
3	06	32	37	07	14289223
3	06	34	26	24	14296728
3	07	30	32	48	14542896
3	07	35	28	40	14563112
3	07	37	31	46	14571502
3	07	39	37	26	14580058
3	08	24	30	29	14780317
3	08	31	33	22	14809174
3	08	32	25	19	14812755
3	08	32	34	12	14813324
3	08	36	39	35	14830051
3	09	24	32	00	15042560
3	09	26	27	37	15050469
3	09	27	26	33	15054497
3	09	34	28	16	15083280
3	10	32	30	34	15337378
3	10	35	25	38	15349350
4	04	55	23	44	18052588
4	05	39	29	20	18249556
4	06	40	27	27	18515675
4	07	26	32	14	18720782
4	07	30	42	03	18737795
4	07	33	32	30	18749470
4	07	39	31	49	18774001
4	07	39	37	08	18774344
4	08	31	24	42	19002922
4	08	32	24	41	19007017
4	08	32	27	45	19007213
4	08	38	34	13	19032205
4	08	41	28	18	19044114
4	08	50	26	21	19080853
4	09	19	21	10	19215690
4	09	25	27	17	19240657
4	09	32	42	31	19270303
4	10	30	44	28	19524380
4	10	35	19	43	19543275
4	10	39	33	39	19560551
4	12	12	31	01	19974081
5	05	24	26	47	22382255
5	07	45	16	25	22991897
5	08	26	32	23	23177239
5	10	19	31	32	23672800
5	10	33	22	15	23729551

Table (c)

A_1	A_2	A_3	A_4	A_5	$\mathcal{N}_{\mathcal{R}}$
0	00	03	00	30	12318
0	03	62	06	02	1040770
2	10	27	27	04	11122372
0	10	03	62	05	2637701
0	00	55	63	60	229372
0	00	06	05	59	24955
0	00	62	09	01	254529
3	06	32	37	07	14289223
0	00	01	53	17	7505
0	00	60	06	24	246168
0	00	02	03	06	8390
0	00	02	05	44	8556
3	07	39	37	26	14580058
0	00	48	57	03	200259
0	00	07	02	57	28857
0	00	00	08	57	569
0	00	04	05	23	16727
3	08	36	39	35	14830051
0	00	51	56	29	212509
0	00	01	59	37	7909
0	00	07	01	47	28783
0	00	62	02	18	254098
3	10	32	30	34	15337378
0	00	02	59	04	11972
0	10	19	62	06	2703238
0	01	00	62	07	266119
0	00	50	04	51	205107
4	07	26	32	14	18720782
0	00	04	09	53	17013
0	00	02	54	27	11675
0	00	00	05	23	343
0	00	55	51	34	228578
4	08	31	24	42	19002922
0	00	00	63	63	4095
0	00	00	03	04	196
0	00	02	58	05	11909
0	00	08	62	03	36739
4	08	50	26	21	19080853
0	00	32	58	53	134837
0	00	06	06	07	24967
0	00	62	01	61	254077
0	00	04	39	15	18895
4	10	35	19	43	19543275
0	00	04	13	60	17276
0	01	36	61	26	413530
0	02	20	53	42	609642
0	00	45	15	62	185342
5	08	26	32	23	23177239
0	01	56	63	09	495561
0	00	13	54	47	56751

Table (d)

Figure 2.2: A relation R and its transformation after domain mapping. Table (a) is the original relation and Table (b) is the resulting table after mapping every attribute value to an integer. Table (c) shows the relation after tuple re-ordering. Table (d) shows the LLVQ coding within blocks.

strings, we may construct a table containing the set of these strings and replace each attribute by an index into the table [6]. Other schemes may be used [7, 13]. Observe that this step by itelf achieves compression because an attribute value that consists of a long string of ASCII characters is mapped to a short number.

3.2 Tuple re-ordering

The next preprocessing step is to re-order the tuples by an ordering rule, such as that defined by φ (Equation 2.2). Table (c) in Figure 2.2 shows the tuples ordered lexicographically by φ. The importance of this step will soon be clear.

3.3 Block partitioning

A problem with conventional data compression techniques is that coding and decoding is performed at the granularity of data objects. In order to restrict the scope of coding/decoding, we partition the re-ordered relation into p disjoint subsets of tuples, $B_1, B_2, \ldots, B_p$. We have chosen the size of a memory page or disk sector as the partition size as it is the unit of I/O transfer. That is, the number of bytes occupied by the set of tuples in a partition is no more than the size of a disk block. When a tuple is required, the block where it resides is transferred from disk to main memory. If tuples in the block are coded, then decoding need only be performed on the block. Hence, coding and decoding is localized.

3.4 Block coding

A block B_k now consists of a set of tuples ordered lexicographically, i.e., $B_k = \langle t_{k,1}, t_{k,2}, \ldots, t_{k,u} \rangle$, $t_{k,i} \in R$, with $t_{k,i} < t_{k,j}$ for $i < j$. The middle tuple in each block B_k is chosen as the representative tuple $\hat{t}_k$ of the block. Thus, every tuple $t_{k,i} \in B_k$ is mapped to $\hat{t}_k$.

Why is the middle tuple representative? After tuple re-ordering and block partitioning, tuples in a block form a *cluster*. The *median* of this cluster is a tuple $\hat{t}$ such that the total distortion $\sum_{i=1}^{u} |\varphi(t_{k,i}) - \varphi(\hat{t})|$ is minimized.

With the representative tuple known, all the other tuples are AVQ coded into pairs as per Definition 2.1. However, the index component is redundant since the representative tuple is known and unique for all tuples in the block. Hence, we need only replace each tuple by its difference from $\hat{t}_k$.

A_1	A_2	A_3	A_4	A_5	$\mathcal{N}_\mathcal{R}$
3	08	32	25	19	14812755
3	08	32	34	12	14813324
3	08	36	39	35	14830051
3	09	24	32	00	15042560
3	09	26	27	37	15050469

Table (a)

A_1	A_2	A_3	A_4	A_5	$\mathcal{N}_\mathcal{R}$
0	00	04	14	16	17296
0	00	04	05	23	16727
3	08	36	39	35	14830051
0	00	51	56	29	212509
0	00	53	52	02	220418

Table (b)

A_1	A_2	A_3	A_4	A_5	$\mathcal{N}_\mathcal{R}$
0	00	00	08	57	569
0	00	04	05	23	16727
3	08	36	39	35	14830051
0	00	51	56	29	212509
0	00	01	59	37	7909

Table (c)

		3	08	57
	2	04	05	23
3	08	36	39	35
	2	51	56	29
	2	01	59	37

Table (d)

Figure 3.3: Stages in coding a block of tuples. Table (a) shows a block of tuples and Table (b) shows the block after LLVQ coding. Table (c) shows the block after subtraction and Table (d) shows the block after run-length coding.

Example 3.2 Consider the fourth block of Table (c) in Figure 2.2 as shown in Table (a) of Figure 3.3. Column $\mathcal{N}_\mathcal{R}$ shows the result of mapping each tuple in column 1 into a number by φ. Taking $\langle 3, 08, 36, 39, 35 \rangle$ as the representative tuple, the other tuples are replaced by their differences. For instance, $\langle 3, 08, 32, 34, 12 \rangle$, which is lexicographically before the representative tuple, is replaced by $\langle 0, 00, 04, 05, 23 \rangle$ since $\varphi(0, 00, 04, 05, 23) = \varphi(3, 08, 36, 39, 35) - \varphi(3, 08, 32, 34, 12) = 14830051 - 14813324 = 16727$. ∎

The differences may be reduced further by making additional subtractions. For a tuple that is lexicographically after the representative tuple, additional difference may be obtained by subtracting the preceding tuple from itself. For a tuple that is lexicographically before the representative tuple, additional difference is obtained by subtracting itself from the succeeding tuple. The following example from Table (c) illustrates:

Example 3.3 Consider tuple $\langle 0, 00, 04, 14, 16 \rangle$ in Table (b). It is replaced by $\langle 0, 00, 00, 08, 57 \rangle$ obtained as follows: $\varphi(0, 00, 00, 08, 57) = \varphi(0, 00, 04, 14, 16) - \varphi(0, 00, 04, 05, 23) = 17296 - 16727 = 569$. This optimization produces Table (c). ∎

Notice the run of leading zeros in each tuple of Table (c). These zeros arise because the differences we are storing require fewer bits than do the original tuples. By coding these runs using run-length coding

[4], we obtain Table (d). The runs are replaced by a count of the number of zeros. Coding is complete when these tuples are concatenated as a single stream of data, with the representative tuple being placed in the front. The stream for the block in the example is:

$$308363935308572040523251562920 15937$$

In AVQ, each block is coded using the above sequence of steps into a stream of bytes. If m is the size of a tuple, the stream may be parsed as follows: The first m bytes give the representative tuple. The next byte is a count field, and gives the number of leading zeros in bytes for the next tuple in the block. If this value is r, then the next $m - r$ bytes are read to get the second tuple. The next byte that follows is again a count field, and the process repeats until all the differences are read. Note that the first and second halves of these differences represent tuples which are lexicographically smaller and larger than the representative tuples respectively.

We end with a note regarding the amount of unused space left in the block after coding the tuples: The number of tuples allocated to a block before coding must be suitably fixed so as to minimize this space. The entire relation R after AVQ is shown in Table (d) of Figure 2.2.

4 DB Structure and Operations

In this section, we consider how access mechanisms may be constructed on the coded tuples, and how the tuples may be retrieved and modified.

4.1 Access method

Figure 4.4 shows an order-3 primary B^+ tree index constructed using the data blocks of Table (d) in Figure 2.2. Each block begins with the representative tuple followed by tuple differences. Notice that the search key in the index is an entire tuple. In conventional primary indices, the search key is usually only an attribute value (primary key).

Operations on the tree-index are performed as usual. Suppose a query wishes to locate the tuple $\langle 4, 07, 39, 37, 08 \rangle$. Starting with the key in the root index node, index node 2 is searched next since it is lexicographically smaller than the root key. There are two search keys in node 2. Following the link corresponding to the smaller of the differences between the tuple and each of the keys, index node 6 is searched. We find again that the second search key is closer to the tuple than the first. This leads us to data block 6,

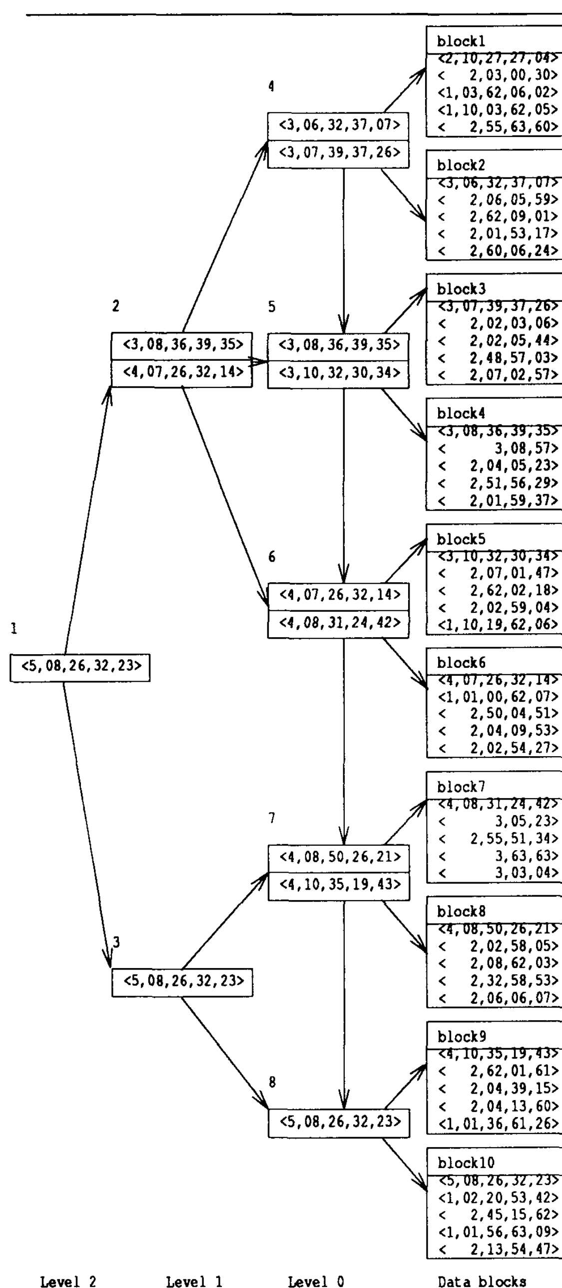

Figure 4.4: Primary index. The search key is an entire tuple. Each block begins with a representative tuple. All tuples following the representative tuple are difference tuples, in which the first value is the count of leading zeros.

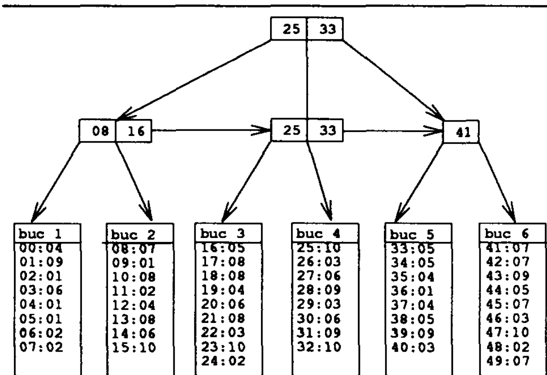

Figure 4.5: Secondary index for A_5. The buckets provide a level of indirection between attributes of A_5 and the tuples of R. Each bucket contains a set of pair $(a : b)$ where a is the attribute value and b indicates the data block where the tuple whose $A_5 = a$ resides.

A_1	A_2	A_3	A_4	A_5	$\mathcal{N}_\mathcal{R}$
3	08	32	25	19	14812755
3	08	32	34	12	14813324
3	08	36	39	35	14830051
3	09	24	32	00	15042560
3	09	26	27	37	15050469

Unquantized block

A_1	A_2	A_3	A_4	A_5	$\mathcal{N}_\mathcal{R}$
0	00	00	08	57	569
0	00	04	05	23	16727
3	08	36	39	35	14830051
0	00	51	56	29	212509
0	00	01	59	37	7909

Quantized block

A_1	A_2	A_3	A_4	A_5	$\mathcal{N}_\mathcal{R}$
3	08	32	25	19	14812755
3	*08*	*32*	*25*	*64*	*14812800*
3	08	32	34	12	14813324
3	08	36	39	35	14830051
3	09	24	32	00	15042560
3	09	26	27	37	15050469

Unquantized block

A_1	A_2	A_3	A_4	A_5	$\mathcal{N}_\mathcal{R}$
0	00	00	00	45	45
0	*00*	*00*	*08*	*12*	*524*
0	00	04	05	23	16727
3	08	36	39	35	14830051
0	00	51	56	29	212509
0	00	01	59	37	7909

Quantized block

Figure 4.6: Tuple insertion in a LLVQ coded block. The two tables above are before insertion, while the two tables below are after insertion. The new tuple is shown in *italic*.

where the tuple resides. This block is now transferred to main memory and decompressed. Thus, traversing the index is the same except that key comparison requires measuring the difference between the key and the target tuple.

When tuples are to be retrieved given certain attribute values only, secondary indices are needed. Figure 4.5 shows an order-3 B^+ tree index where A_5 is the search key. Since the relation is physically clustered via φ, the index is non-clustering and secondary. This explains the extra level of indirection provided by the buckets in the figure. Each bucket contains a pair $(a : b)$ where b indicates the data block whose tuples have $A_5 = a$. Suppose we wish to execute $\sigma_{A_5=34}(R)$. Traversing the index points to bucket 5, where the tuple resides.

4.2 Tuple insertion and deletion

How are tuple insertions and deletions supported in a compressed database? Suppose we wish to insert tuple $\langle 3, 08, 32, 25, 64 \rangle$. Using the primary index, we identify data block 4 as the set of insertion. The tuple is found to lie lexicographically between the first and second tuple in the block. Thus, the differences from the representative tuple must be recomputed. Figure 4.6 shows the result of tuple insertion.

Notice that differences are re-computed only for tuples before the representative tuple, and that the changes are confined to the affected block. For tuple deletion, the primary index is similarly used to locate the data block, and changes made within the block. Tuple modification may simply be defined as a combination of tuple insertion and deletion.

In summary, standard database operations remain the same even when the database is AVQ coded. The only difference is that the search key of the primary index of a AVQ coded relation is an entire tuple. All other indices are non-clustering and secondary, as in standard databases. A further advantage of an AVQ-coded database is that the storage requirements for the indices will be reduced because the number of data blocks for storing the database has been reduced by compression. Although we have illustrated the use of tree indices as the access mechanisms, we do not preclude the use of other methods, such as hashing.

5 Performance Evaluation

The goals of database compression are both to reduce space requirements as well as to improve the response time of I/O intensive queries. In this section, we look into the compression ratio (Section 5.1), average compression/decompression time (Section 5.2) and the effects of database compression on query response time (Section 5.3). We shall see how both the reduction of I/O and the improvement in I/O bandwidth contribute to the improvement in query response time.

Test number	1	2	3	4
Data skew	Yes	Yes	No	No
Domain variance	Small	Large	Small	Large

Table (a) Test characteristics

No. of tuples	Test 1	Test 2	Test 3	Test 4
10^4	73.0%	65.6%	73.0%	65.6%
10^5	70.6%	66.7%	70.6%	66.7%
10^6	71.4%	65.5%	71.4%	65.5%

Table (b) Percentage reduction in size

Figure 5.7: Compression efficiency. The figures in Table (b) are obtained via the formula: $100(1-a/b)\%$ where b and a are the size of the database before and after coding respectively.

5.1 Compression efficiency

In order to compare the compression performance of each of the variants, we only have to compare the size of a relation before and after compression. However, what constitutes a *typical* relation?

In order to ensure a fair evaluation, we generated relations of various sizes and characteristics. They differed in: (1) relation size (i.e., the number of tuples), (2) variance in attribute domain size, and (3) attribute value skew. When the differences in domain sizes were no more than 10% of the average domain size, we took the domain size variance to be low. When the differences were more than 100%, we took the variance to be high. The distribution of values within a domain was taken to be skewed when 60% of the values were drawn from 40% of the domain. When no skew existed, values were drawn uniformly from the domain. The number of attribute domains of all relations were fixed at 15. We measured the number of disk blocks required by a relation under these variants.

With these parameter variations, four sets of simulations were performed. The domain variance and attribute value skew parameters give a total of four combinations of relation characteristics: small variance and no data skew, large variance and no data skew, small variance and data skew, large variance and data skew. The relation size are varied in each of these combinations. These combinations are tabulated in Table (a) in Figure 5.7. The results of the simulations are shown in Figure 5.7. The following observations may be made:

- The data size is greatly reduced for a compressed relation. This is clear from the high compression efficiencies shown in Table (b). Recall that the relation being compressed is a table of *numerical* tuples. Considering the domain mapping already performed, the actual efficiency is higher.
- Homogeneity in domain sizes affects the compression efficiency. More homogeneity increases efficiency, as the figures in Tests 1 and 3 are relatively higher than the figures in Tests 2 and 4. Therefore, a relation whose range of actual attribute values in each domain does not differ much yields better compressibility.
- Data skew does not seem to affect compression efficiency as the figures in Tests 1 and 2 are the same as the figures in Tests 3 and 4.

5.2 Coding/decoding time overhead

We measure the average time taken to encode a set of tuples such that the size of the coded tuples can be allocated to a disk block with minimal unused space left in the block. We also measure the time to decode the block.

The relation characteristics are as follows: We use a relation with 16 attributes of varying domain sizes. After domain mapping, each tuple is 38 bytes and there are 10^5 tuples in the relation. The block size is taken to be 8192 bytes.

The measurements are made for each of the three techniques. For each of them, we perform the coding 100 times, and then the decoding 100 times. The average times for each operation are then computed. Before coding, the required number of tuples is first loaded into main memory so as to offset any I/O time. The measurements are taken when the coding routine is the only user-level process executing in the system. The results are shown in rows 1 and 2 in Figure 5.9.

It is to be noted that the block after decoding is a collection of tuples whose attribute values are integers. Another level of decoding is needed to map the values back to their alphanumeric values originally. We have omitted the measurement of this because the decoding overhead is *approximately* the same for all techniques.

5.3 Response time

In order to perform any measurement, we need the notion of a typical query. This is difficult because as there are many possibilities. Each query is specified by (1) the number of attributes involved, (2) the logical operators on these attributes, (3) the arithmetic operations to be performed, etc. To simplify things, we make the following assumptions:

- Queries are I/O-intensive, so that they are directly affected by the I/O bottleneck problem.
- All queries reduce to a set of tuple access operations.
- The time for these operations form the bulk of the overall query response time. Thus, it directly affects query performance.

We consider query of the form $\sigma_{a \leq A_k \leq b}(R)$, where A_k is any non-primary key attribute and $a, b \in A_k$. By varying a and b suitably, the number of tuples accessed can be made larger, and thus more I/O-intensive. The tuple access operation is the only one in the query and directly determines the cost of the query.

C_1, the total time taken to bring in the relevant disk blocks into main memory for further processing in the above query is given by the following expression:

$$C_1 = I + N(t_1 + t_2) \quad (5.7)$$

where I is the index search time, N is the number of disk blocks accessed, t_1 is the time to read a block, and t_2 is the decompression time per block. When the database is not compressed, the corresponding cost C_2, is:

$$C_2 = I + N(t_1 + t_3) \quad (5.8)$$

where t_3 is the time to read and extract a block into a set of tuples. This time is included in t_2, since the decompression yields a set of tuples.

5.3.1 Estimating I

I, the time required to search the access mechanisms (indices) to locate the block where the desired tuples reside, is likely to be a relatively small component in comparison with t_1. It is dominated by the I/O needed to bring in the small number of index blocks. Assuming the number of secondary index blocks to be 5% of the total number of data blocks, which is 189 and 64 respectively for the uncoded and coded relation. The value of I is shown in rows 5 and 6 in Figure 5.9.

5.3.2 Estimating t_1, t_2, t_3

t_1, the average I/O time per disk block is estimated as follows: The components of an average disk I/O read/write are: *seek time*, *rotational delay*, *data transfer time* and *controller overhead*. Seek time, rotational delay and controller overhead are usually in the range of 10–20 ms, 8 ms, and 2 ms respectively [8]. Assuming a data transfer rate of 3 Mb/sec, the average I/O time for a block size of 8192 bytes is:

Attribute No.	1	2	3	4	5	6	7
No coding	33	189	189	189	184	105	183
AVQ	1	64	64	64	64	55	64

Attribute No.	8	9	10	11	12	13	14	15
No coding	151	189	189	161	189	11	189	1
AVQ	64	64	64	64	64	11	64	1

Figure 5.8: Estimating N, the number of blocks accessed.

20 ms + 8 ms + (8192 b/3 Mb) ms + 2 ms $\approx$ 30 ms
As the relation characteristics are the same as that of Section 5.2, the average time for single block decompression, t_2, is already measured in that section. The estimations for t_3 are given in row 4 in Figure 5.9.

5.3.3 Estimating N

We measure N via simulations. The relation R used has the same characteristics as that of Section 5.2. The selection query $\sigma_{a \leq A_k \leq b}(R)$ has three *parameters*: k, a, b. Figure 5.8 gives the number of blocks accessed when executing the query for each of the attributes of a tuple, i.e., $k = 1, 2, \ldots, 15$, and where $a = 0.5 \times |A_k|$.

Observe that only one block is accessed when $k = 15$ because A_{15} is the primary key. The number of blocks accessed on average is computed from these figures and shown in rows 7 and 8 in Figure 5.9. AVQ reduces the number of blocks accessed by $100(1 - 55/153.6) = 64.2\%$.

5.3.4 Results

Given the relation (Section 5.2) and query (Section 5.3), Figure 5.9 shows the results of combining all the components of the total time taken to bring in the relevant disk blocks into main memory for the cases when the relation is compressed (C_1) and when the relation is uncompressed (C_2). For instance, the query I/O time of an uncoded relation on the HP 9000/735 is $153.6(30 + 1.34) = 4.81$ secs, and that of a coded relation is $55(30 + 13.85) = 2.41$ secs.

AVQ shows improvements which are likely to increase with processor technology, as the faster machines show higher ratios. Processor technology progresses at a faster rate than disk technology. Thus, the t_2 component is likely to decrease, with t_1 staying about the same.

No.	Description	HP 9000/735	Sun 4/50	Dec 5000/120
1	Block coding time (msec)	13.91	40.29	69.92
2	Block decoding time (msec), t_2	13.85	40.45	61.33
3	Single block I/O time (msec), t_1	30.00	30.00	30.00
4	Time to extract tuples (msec), t_3	1.34	3.70	9.77
5	Index search time (uncoded) (sec), I	0.283	0.283	0.283
6	Index search time (AVQ-coded) (sec), I	0.096	0.096	0.096
7	No. of blocks accessed (uncoded), N	153.6	153.6	153.6
8	No. of blocks accessed (AVQ-coded), N	55.0	55.0	55.0
9	Total I/O time (uncoded) (sec), C_2	5.093	6.013	6.403
10	Total I/O time (AVQ-coded) (sec), C_1	2.506	3.966	5.116
11	Improvement	50.8%	34.0%	20.1%

Figure 5.9: **Response time improvements. The figures in the table are determined in the previous sections. The percentage response time savings in row 11 are computed using the formula:** $100(1 - C_1/C_2)\%$.

6 Conclusions

The motivation for this work is the I/O bottleneck problem caused by the ever-increasing disparity between cpu/memory and disk speeds. Adopting the data compression approach, we have presented a compression technique tailored specifically for relational databases. AVQ is based on vector quantization and is a lossless version that also supports standard database operations.

AVQ does not incur some of the computational overheads of conventional VQ. The output vectors are computed without resorting to any codebook computation algorithms. There is no need for codewords as the each vector is associated with a disk block, and no searching of the codebook is necessary. These features make AVQ more computationally efficient than conventional VQ in terms of coding and decoding.

References

[1] R. AGRAWAL, D. J. DEWITT. Whither Hundreds of Processors in a Database Machine? *Proceedings of the International Workshop on High-Level Architectures*, 1984.

[2] M. A. BASSIOUNI. Data Compression in Scientifc and Statistical Databases. *IEEE Transactions on Software Engineering*, Vol. 11, No. 10, pp. 1047–1058, October 1985.

[3] A. GERSHO, V. CUPERMAN. Vector Quantization: A Pattern-Matching Technique for Speech Coding. *IEEE Communications Magazine*, Vol. 21, pp. 15–21, December 1983.

[4] S. W. GOLOMB. Run-Length Encodings. *IEEE Transactions on Information Theory*, Vol. 12, pp. 399–401, Jul. 1966.

[5] R. M. GRAY. Vector Quantization. *IEEE ASSP Magazine*, Vol. 1, pp. 4–29, April 1984.

[6] G. GRAEFE, L. D. SHAPIRO. Data Compression and Database Performance. *Proceedings of the ACM/IEEE-Computer Society Symposium on Applied Computing*, Kansas City, Montana, April 1991.

[7] B. HAHN. A New Technique for Compression and Storage of Data. *Communications of the ACM*, Vol. 17, No. 8, pp. 434–436, August 1974.

[8] R. H. KATZ, G. A. GIBSON, D. A. PATTERSON. Disk System Architectures for High Performance Computing. *Proceedings of the IEEE*, Vol. 77, No. 12, pp. 1842–1858, December 1989.

[9] Y. LINDE, A. BUZO, R. M. GRAY. An Algorithm for Vector Quantizer Design. *IEEE Transactions on Communications*, Vol. 28, No. 1, pp. 84–95, January 1980.

[10] W. K. NG, C. V. RAVISHANKAR. A Physical Storage Model for Efficient Statistical Query Processing. *Proceedings of the 7th International Working Conference on Statistical and Scientific Databases*, pp. 97–106, Charlottesville, Virginia, September 1994.

[11] W. K. NG, C. V. RAVISHANKAR. Tuple Differential Coding. U.S. Patent pending, 1994.

[12] W. K. NG, C. V. RAVISHANKAR. A Tuple Model for Summary Data Management. *Proceedings of the 6th International Conference on Management of Data*, Bangalore, India, December 1994.

[13] H. K. T. WONG, H. F. LIU, F. OLKEN, D. ROTEM, L. WONG. Bit Transposed Files. *Proceedings of the International Conference on Very Large Data Bases*, pp. 448-457, 1985.