Block-Oriented Compression Techniques for Large Statistical Databases

Wee Keong Ng, *Member*, *IEEE*,
and Chinya V. Ravishankar, *Member*, *IEEE*

Abstract—Disk I/O has long been a performance bottleneck for very large databases. Database compression can be used to reduce disk I/O bandwidth requirements for large data transfers. In this paper, we explore the compression of large statistical databases and propose techniques for organizing the compressed data such that standard database operations such as retrievals, inserts, deletes and modifications are supported. We examine the applicability and performance of three methods. Two of these are adaptations of existing methods, but the third, called Tuple Differential Coding (TDC) [16], is a new method that allows conventional access mechanisms to be used with the compressed data to provide efficient access. We demonstrate how the performance of queries that involve large data transfers can be improved with these database compression techniques.

Index Terms—Database compression, data compression, physical organization, statistical database.

1 Introduction

The storage of very large databases may constitute a significant portion of the cost of managing them. The compression of data therefore becomes important as the amount of data grows. In addition, database management systems must provide efficient access to the compressed data. In this paper, we explore the compression of large statistical databases, and propose several techniques for organizing the compressed data such that standard database operations are supported.

The term *statistical database* is used here as a generic term denoting databases holding information amenable to statistical analysis. Examples are databases constructed from social, economic, inventory, environmental, or demographic surveys or experiments. These statistical databases have the following characteristics:

1) They are usually large and of indefinite retention. For instance, the 1990 5-percent Public Use Microdata Samples (PUMS) from the U.S. Bureau of Census [19] is about 4 gigabytes. These data are never erased, and as more censuses are conducted in the future, the total database size will only grow.
2) By their nature, statistical database queries generate a lot of I/O. These queries usually access a large portion of several related databases in order to consolidate and summarize information for the users. They are also aggregational in that arithmetic operations are usually performed to compute statistics such as means and variance on selected fields.
3) They are similar to conventional relational databases in that each data set (relation) is a collection of records (tuples) with a fixed number of fields (attributes). However, there are two major differences. First, the data set may not be normalized and there may be no primary keys. Records in such cases are instances in a survey, an experiment, or a simulation, and are characterized by parameters or categories (attributes) without necessarily having any unique identifications [24]. Second, the attribute domains are simpler. They are discrete and of finite size because the database is usually compiled from surveys or experiments with a set of questions (attributes). The domain is usually encoded so that each attribute value corresponds to one of a designated set of answers to the question.
4) The database is usually available in raw form as one or more flat files of ASCII characters, with records stored one per line contiguously. This approach permits easy data distribution and access without relying on complicated data formats. (See Section 6.1 for a snapshot of the census database.)
5) There is clustering throughout the database in the sense that many records will often have identical values for certain attributes. Such clustering is indicative of information redundancy that may be exploited during compression.
6) Statistical databases are considerably more stable than ordinary databases—such as financial or airline reservation databases—since record updates are very rare. However, appending new records is a common operation.

The first two characteristics confirm that statistical databases are constrained by disk I/O performance. Based on the other characteristics, we propose several compression techniques, which preserve record identity within the compressed data, so that individual records may be manipulated as if the database were uncompressed.

• *W.K. Ng is with the School of Applied Science, Nanyang Technological University, Singapore. E-mail: wkn@sentosa.sas.ntu.ac.sg.*
• *C.V. Ravishankar is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109. E-mail: ravi@eecs.umich.edu.*

Manuscript received Feb. 10, 1995; revised Oct. 24, 1995.
For information on obtaining reprints of this article, please send e-mail to: transkde@computer.org, and reference IEEECS Log Number K97017.

We shall rely on relational database terminology in our expositions. A statistical database is relational in its structure, but it may not satisfy the normal forms. We shall use terms like relation, tuple, attribute, to refer to data set, record, field respectively. We shall also rely on relational algebraic operators such as selection (σ) and projection (π).

1.1 Organization of Paper

Section 2 examines the difference between *data* and *database* compression. We argue that database compression is *fundamentally* different from data compression. Thus, conventional data compression techniques are not directly or immediately applicable to database compression. This comparison allows us to establish a set of requirements for database compression techniques. Based on these requirements, we design three database compression techniques in Section 3.

Section 4 is concerned with issues relating to the practical implementation of the proposed compression techniques. We describe how standard database operations may continue to be supported in a compressed database.

One must evaluate the performance of any proposed algorithm with respect to its design objectives. In Section 5, we compare the compression efficiencies of the proposed compression techniques and evaluate their impact on the response time of queries. We show that the techniques reduce the response time by I/O reduction. Section 6 discusses related work in the area of database compression. Finally, Section 7 concludes the paper.

2 DATABASE COMPRESSION VERSUS DATA COMPRESSION

The requirements of database compression are very different from those of data compression in general. First, because database compression must be inherently lossless, we are interested only in lossless techniques, which allow the original data to be fully recovered from its compressed form. However, as we argue below, this is not the only, or even primary difference. Database compression techniques must not hinder operations on the database, and we argue that conventional compression methods are quite inadequate from this point of view.

Current techniques for lossless data compression may be categorized into two broad classes:

1) *statistical* techniques and
2) textual substitution techniques.[1]

Statistical techniques separate the work of compression at the source into two parts: statistical *data modeling* and *coding* [20], [28]. Statistical data modeling aims to capture the frequency of occurrence of *source words* in the data stream. These frequencies allow different code words to be assigned to different source words. The data stream is usually coded using *arithmetic coding* [13], using these frequencies. As arithmetic coding has been shown to be optimal with respect to a given set of frequencies [13], most statistical techniques strive to produce as accurate a statistical model of the source data stream as possible.

The class of Lempel-Ziv techniques, which we refer to as the LZ techniques are typical of textual substitution methods. These techniques achieve compression by replacing strings of symbols with pointers to previous occurrences of the same strings. All techniques in this class are variants of [30], [31].

Database compression is fundamentally different from data compression. Conventional data compression techniques, such as the above, tacitly adopt a model of compression patterned after Shannon's model of communication [23], which consists of an abstract channel through which a source generates an infinite sequence of data symbols to a destination (see Fig. 1). The transmitter and receiver share a statistical *data model* which provides information and knowledge to the coder and decoder respectively. A statistical data model contains a set of word-frequency pair for each word (a sequence of symbols) appearing in the source. Thus the shared model captures the characteristics of the source, and the efficiency of compression depends upon how completely and faithfully source characteristics are captured. Source word frequencies are a primary input to the statistical data model. When such frequencies are not known statically, the statistical data model may be updated dynamically as more information becomes available about the source.

There are two important characteristics of the compression model shown in Fig. 1:

1) *Data access mode*: Data is processed *serially*, as indicated by the FIFO data store between the transmitter and receiver. The FIFO representation is purely conceptual, and does not require a physical store between the transmitter and receiver. Source data is serially encoded, and serially decoded either *on-line* or *off-line* by the receiver. In the on-line mode, the transmitter and receiver form a FIFO producer-consumer pair, and data flows from the source to the destination in a pipelined fashion. In the off-line mode, data is encoded in its entirety and stored before being decoded serially in its entirety by the receiver. In both cases, data is decoded *in the order* it was encoded.
2) *Statistical data model consistency*: The second characteristic is that the statistical data models maintained within the transmitter/receiver must be *consistent* with each other: If the same source data were to be compressed again with the same statistical data model, it should yield the same encoded data. Likewise, if the same encoded data were to be decoded with the statistical data model, it should yield the same original data.

These two properties conflict with the requirements of database compression. To be useful, a database compression technique must allow standard database operations such as tuple access, insertion/deletion and modification over any part of the database. In other words, a database

1. Textual substitution may be seen as a statistical technique [6], but we distinguish between the two here primarily because of their different historical roots.

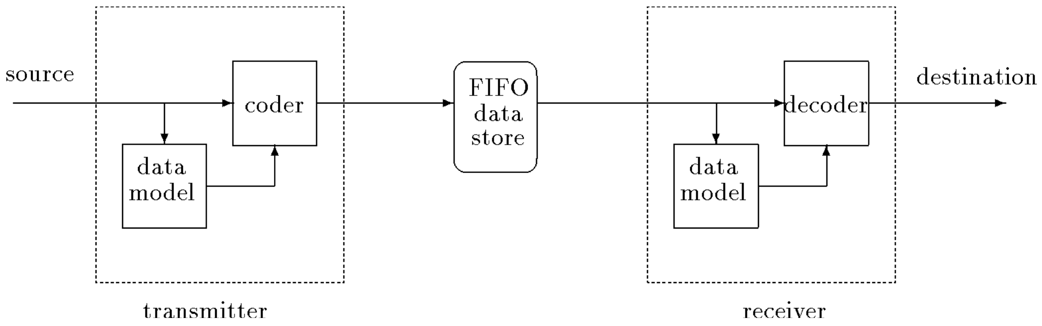


Fig. 1. Conventional data compression model. This model is a reflection of the *modern paradigm* of data compression. An important component of the model is the FIFO data store between the transmitter and the receiver.

compression technique should support *random, localized* (non-FIFO) access to a compressed database. This is impossible to achieve in any method based on the model in Fig. 1 due to the *data-model consistency* problem. The statistical data models in the transmitter and receiver must be consistent with each other at all times, but the random nature of access and update means that the statistical characteristics of the data change with time. Thus, different parts of the database may have been encoded with different statistical data models, since it is clearly impractical to recode the entire database during tuple updates in order to preserve consistency. This consistency requirement restricts the scope of applicability of many compression techniques that rely heavily on data modeling, be they statistical or LZ-based.

The conventional and database models of data compression must address different modus operandi. The standard categories of data compression techniques assume a model of compression that is not directly suitable for database compression. The design of a database compression technique requires a fundamentally different approach.

This paper presents TDC, a novel database compression method based on a fundamentally different approach. Consider the typical model used for coding, as shown in Fig. 1. The data source is generally taken to be a producer of a stream of signals or bits, and compression must be performed on bits as they are generated, to preserve signal integrity. That means that the ordering of source information is determined by the order of generation. Information ordering is important because it also determines the statistical data model for compression.

We argue in this paper that this is a needlessly restrictive view from the database perspective. Database compression can exploit record reordering to improve compression over standard data compressors. With a suitable ordering scheme, the order in which information in a database is generated becomes irrelevant; only the statically defined ordering is significant. This is a fundamental departure from the traditional approach.

3 Techniques for Database Compression

In this section, we discuss three database compression techniques:

1) Bit compression (BIT),
2) Adaptive Text Substitution (ATS), and
3) Tuple Differential Coding (TDC) [16].

The first two are existing methods, but TDC is a new method we have developed specifically with the requirements of statistical databases in mind. In order to accommodate the standard operations on a database, a database compression technique should exhibit the following features:

1) Tuple access should not require massive compression and decompression. One would not use a compression technique that decompresses and recompresses the entire database every time a tuple is accessed. Thus, the scope of compression should be reduced. A natural choice for all disk-based database systems is a disk sector or block, the unit of a disk I/O operation. When a tuple is desired in a query, only the block where it resides is brought into memory, where it is decompressed. Hence, the scope of decompression is reduced to a block. As the amount of achievable compression on a volume of data depends on the amount of redundancy in the data, the basic trade-off in choosing the granularity of a scope is that smaller scope permits faster decompression but offers lower achievable compression.
2) It should provide localized access to compressed tuples. One must be able to build access mechanisms on a compressed database. Reducing the scope of compression helps achieve this goal because a tuple is now addressed by the block where it resides. Conventional access mechanisms such as B-trees can easily be constructed to access tuples in blocks.
3) Compression and decompression should be fast enough so as not to offset its advantages, i.e., a database compression technique should not be so complex as to offset its space and bandwidth reduction advantages. Although the scope of compression is reduced to a block, it is still critical that the decompression time be less than the time to transfer the uncompressed block from disk to memory. The speed of compression / decompression and the compression efficiency should be balanced carefully.

A_1	A_2	A_3	A_4	A_5
production	*manager*	*D*	*24*	*40*
marketing	*supervisor*	*B*	*35*	*41*
marketing	*manager*	*C*	*30*	*38*
marketing	*part-time*	*B*	*25*	*39*
personnel	*supervisor*	*C*	*34*	*50*
marketing	*worker*	*C*	*30*	*45*
personnel	*manager*	*A*	*35*	*60*
personnel	*worker*	*C*	*50*	*46*
production	*worker*	*B*	*30*	*42*
management	*manager*	*B*	*20*	*10*
production	*part-time*	*D*	*40*	*38*
marketing	*worker*	*D*	*43*	*27*
personnel	*worker*	*B*	*35*	*29*
marketing	*manager*	*B*	*25*	*39*
production	*manager*	*A*	*20*	*50*
management	*part-time*	*A*	*32*	*30*
production	*part-time*	*C*	*30*	*28*
personnel	*supervisor*	*D*	*20*	*47*
personnel	*manager*	*D*	*20*	*33*
personnel	*part-time*	*D*	*25*	*23*
marketing	*supervisor*	*A*	*30*	*50*
management	*worker*	*C*	*20*	*34*
management	*manager*	*A*	*40*	*40*
production	*worker*	*A*	*22*	*41*
marketing	*manager*	*A*	*32*	*38*
personnel	*manager*	*C*	*28*	*34*
personnel	*part-time*	*B*	*30*	*36*
production	*part-time*	*A*	*35*	*48*
production	*supervisor*	*A*	*30*	*49*
management	*worker*	*B*	*35*	*38*
management	*supervisor*	*C*	*32*	*39*
marketing	*manager*	*D*	*25*	*40*
personnel	*worker*	*A*	*22*	*41*
production	*worker*	*D*	*25*	*56*
marketing	*part-time*	*D*	*20*	*40*
personnel	*supervisor*	*B*	*20*	*52*
management	*worker*	*D*	*45*	*33*
marketing	*part-time*	*A*	*20*	*40*
personnel	*part-time*	*A*	*30*	*41*
production	*supervisor*	*C*	*35*	*40*

Table (a)

A_1	A_2	A_3	A_4	A_5
1	1	2	24	40
2	0	1	35	41
2	1	3	30	38
2	3	1	25	39
3	0	3	34	50
2	2	3	30	45
3	1	0	35	60
3	2	3	50	46
1	2	1	30	42
0	1	1	20	10
1	3	2	40	38
2	2	2	43	27
3	2	1	35	29
2	1	1	25	39
1	1	0	20	50
0	3	0	32	30
1	3	3	30	28
3	0	2	20	47
3	1	2	20	33
3	3	2	25	23
2	0	0	30	50
0	2	3	20	34
0	1	0	40	40
1	2	0	22	41
2	1	0	32	38
3	1	3	28	34
3	3	1	30	36
1	3	0	35	48
1	0	0	30	49
0	2	1	35	38
0	0	3	32	39
2	1	2	25	40
3	2	0	22	41
1	2	2	25	56
2	3	2	20	40
3	0	1	20	52
0	2	2	45	33
2	3	0	20	40
3	3	0	30	41
1	0	3	35	40

Table (b)

Fig. 2. A relation *R* and its transformation after domain mapping. Table (a) shows the raw form in which a statistical data set is usually available. Every attribute value is mapped onto an integer.

4) The tuple structure of a relation should be preserved. One would like to be able to access each tuple individually.

In this paper, we present three block-based database compression techniques. Two of them, BIT and ATS, are adaptations of conventional data compression techniques. The third one, TDC, exploits the redundancy among tuples differently to achieve compression.

Throughout this section, we shall be using the relation in the following example to illustrate the concepts involved.

EXAMPLE 1. Table (a) in Fig. 2 shows a relation *R* with five attribute domains A_1, A_2, A_3, A_4, A_5 denoting the **department**, **job title**, **insurance grade**, **income in thousands**, and **hours worked per week** attributes, respectively. The size of each domain, i.e., the number of different attribute values, is 4, 4, 4, 64, 64, respectively. Table (b) in this figure shows the same relation with all attribute values mapped to numbers. This is usually the raw form in which a statistical data set is available; i.e., a file of numerals corresponding to contiguous records. We preserve tuple identity by displaying them as individual tuples. The relation in the figure has been partitioned into *blocks*. Each compression technique compresses a block individually.

3.1 Bit Compression (BIT)

Bit compression (BIT) or compaction is a well-known technique. If a (numerical) attribute domain has size k containing values 0 to $k - 1$, the number of ASCII characters required to represent each attribute value is $\lceil \log_{10} k \rceil$. This requires $8\lceil \log_{10} k \rceil$ bits, assuming each character is a byte of 8 bits. However, we may represent each attribute value in $\lceil \log_2 k \rceil$ bits, yielding a savings of $8\lceil \log_{10} k \rceil - \lceil \log_2 k \rceil$ bits. For example, let $A = \{0, 1, \ldots, 899\}$ be an attribute domain where each attribute value requires $\lceil \log_{10} 900 \rceil = 3$ ASCII characters or equivalently, 24 bits. Since there are 900 different attribute values, we may represent each value in $\lceil \log_2 900 \rceil = 10$ bits, thus yielding a savings of 14 bits. Therefore, the idea of BIT is to compact every attribute value so that the entire tuple is bit-compressed. This technique is simple and fast; however it does not yield high compression.

EXAMPLE 2. Let us see how BIT is applied to the relation in Example 1. Consider the first tuple $\langle 1, 1, 2, 24, 40\rangle$, which requires 7 ASCII characters or 56 bits to store. Its binary representation is $\langle 01_2, 01_2, 10_2, 011000_2, 101000_2\rangle$. After compaction, the tuple becomes 0001 0110 0110 0010 1000 or 16,628 in hexadecimal, and requires only 18 bits or under 3 bytes to store. Bit compression is applied systematically to all tuples of a block.

3.2 Adaptive Text Substitution (ATS)

ATS is an adaptive text substitution technique by Welch [26], and an improvement over the Ziv and Lempel technique described in [31]. The technique works as follows: If a sequence of symbols has occurred previously, replace it by a pointer to that previous occurrence. This is basic text substitution. Variants of this basic technique alter the scope of backward reference, i.e., how far back the technique may go to look for matching string of symbols. The collection of pointers form the dictionary, which is built up dynamically at the same time that text substitution is being performed. Hence, the term adaptive. This technique is also employed in the widely used Unix `compress` utility. Here, we apply ATS to compress and decompress a block of tuples. Thus, the scope of backward reference of the technique is reduced.

EXAMPLE 3. Consider the first block consisting of tuples $\langle 1, 1, 2, 24, 40\rangle$, $\langle 2, 0, 1, 35, 41\rangle$, $\langle 2, 1, 3, 30, 38\rangle$, and $\langle 2, 3, 1, 25, 39\rangle$ in Fig. 2. The block is presented to ATS as the stream of symbols 1122440201354121330382312539 after concatenating all the tuples. The result of compression is a stream of bytes requiring less storage space than the input stream of symbols. We have omitted the details of how the output stream is derived, as the technique is well documented in [26]. In order to minimize the amount of unused space in the block, the original set of tuples can be appropriately increased so that the compressed stream of bytes leave minimal unused space in the block.

3.3 Tuple Differential Coding (TDC)

As we shall illustrate in Section 5, BIT and ATS do not compress well. In this section, we discuss a new technique called Tuple Differential Coding (TDC) which we have recently introduced in [15], [16], [17], [18]. As it is one of the main contributions of this paper, we shall present a more formal and detailed description of the technique.

3.3.1 Motivation

A relation R may be perceived geometrically as a set points in an n-dimensional space (as made precise in Section 3.3.2). As a consequence of characteristic 5 (see Section 1) of statistical databases, tuples that share certain attribute values in R form clusters in this space. For instance, in an employee relation, one will typically find many employee tuples sharing the same values for the department and job-title attributes. Such clusters are indicative of a form of redundancy that we can exploit to reduce the storage requirements of the tuples. For example, many image compression techniques exploit redundancy that is present when adjacent elements are correlated.

Instead of storing tuples explicitly in tabular form as conventional databases do, one may capture and store the *differences* among them. If these differences require less space for storage on average than the original tuples, compression is achieved. The idea is elaborated below.

3.3.2 Definitions

A relation scheme $\mathcal{R} = \langle A_1, A_2, \ldots, A_n\rangle$ is a sequence of attribute domains where $A_i = \{0, 1, \ldots, |A_i| - 1\}$ for $1 \le i \le n$. (Note that A_i is taken to be a set of integers because of characteristic 3 (see Section 1) of a statistical database.) $\mathcal{R}$ may also be viewed as an n-dimensional space composed of tuples from the cartesian product of the sequence of attributes, $A_1 \times A_2 \times \cdots \times A_n$, i.e., a tuple $a_1 \times a_2 \times \cdots \times a_n$ in $\mathcal{R}$, where $a_i \in A_i$ for $1 \le i \le n$, is a point in the space.

3.3.3 Step 1: Tuple Reordering

All points (or tuples) in $\mathcal{R}$ may be totally ordered via an ordering rule. An example is the *lexicographical* order with respect to the attribute sequence in $\mathcal{R}$ defined by function $\varphi\colon \mathcal{R} \to \mathcal{N}_{\mathcal{R}}$, where $\mathcal{N}_{\mathcal{R}} = \{0, 1, \ldots, \|\mathcal{R}\| - 1\}$ and $\|\mathcal{R}\| = \prod_{i=1}^{n} |A_i|$:

$$\varphi(a_1, a_2, \ldots, a_n) = \sum_{i=1}^{n}\left(a_i \prod_{j=i+1}^{n} |A_j|\right) \tag{3.1}$$

for all tuples $\langle a_1, a_2, \cdots a_n\rangle \in \mathcal{R}$. (Note that 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.) The inverse of φ is defined as:

$$\varphi^{-1}(e) = \langle a'_1, a'_2, \ldots, a'_n\rangle \tag{3.2}$$

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

$$a'_i = \left\lfloor \frac{a^r_{i-1}}{\prod_{j=i+1}^{n} |A_j|} \right\rfloor \tag{3.3}$$

$$a^r_i = a^r_{i-1} - a'_i \prod_{j=i+1}^{n} |A_j| \tag{3.4}$$

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

Given a tuple $t \in \mathcal{R}$, φ converts it to a unique integer $\varphi(t)$ that represents its *ordinal* position within 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 $\varphi(t_i) < \varphi(t_j)$.

3.3.4 Step 2: Attribute Domain Ranking

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

Given a relation $R \in \mathcal{R}$, choosing the optimal domain ordering that yields the best achievable compression is NP-complete. It can be shown that the Optimal Linear Arrangement problem, which is known to be NP-complete [10], can be reduced to the optimal domain ordering problem. Due to limitations in the length of the paper, we shall omit the proof. Nonetheless, there are a few heuristics for choosing a good but suboptimal domain ordering. One heuristic is to rank them by the frequency of use, i.e., the more frequently used (in queries) attribute domains are ranked lower (positioned to the left) than the less frequently used domains. Another heuristic is to award a higher rank to domains whose total unique attribute values in the relation is large. For example, a domain that is a candidate key of the relation has attribute values that appear exactly once for each tuple in the relation. This domain should be given a higher rank because there is no duplication of attribute values in the relation.

Table (a) in Fig. 3 shows the tuples from Fig. 2 ordered lexicographically by φ with respect to the attribute sequence A_1, A_2, A_3, A_5, A_4. (The first row in Fig. 2 is now the 11th row in Fig. 3, Table (a).) The attributes have been reordered under the permutation τ defined as:

$$\tau = \begin{pmatrix} 1\,2\,3\,4\,5 \\ 1\,2\,3\,5\,4 \end{pmatrix}.$$

Column $\mathcal{N}_{\mathcal{R}}$ shows the ordinal numbers of the corresponding tuples.

3.3.5 Step 3: Block Partitioning

We next partition the reordered relation into disjoint blocks (subsets) of tuples. 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 compressed, then decompression need only be performed on the block. The block partitions in Fig. 3 are shown by the line demarcations.

3.3.6 Step 4: Block Encoding

A block now consists of a set of tuples ordered lexicographically. Using the first tuple as a reference, each succeeding tuple is replaced by its difference (in ordinals) with respect to its preceding tuple. Consider block 1 of Table (b) in Fig. 3 The first difference after the first tuple is

$$\begin{aligned} 4{,}168 &= \varphi\,(0, 0, 1, 01, 08) \\ &= \varphi\,(0, 0, 3, 39, 32) - \varphi\,(0, 1, 1, 40, 40) \\ &= 18{,}984 - 14{,}816 \end{aligned}$$

In general, if t_i and t_j are consecutive tuples in block k of Table (a), then the entry in Table (b) corresponding to t_j is $\varphi^{-1}(\varphi(t_j) - \varphi(t_i))$.

Since the differences are numerically smaller than the tuples, they require fewer bytes of storage. We encode the variable-size differences by using *run-length coding* [11], [25] to encode the number of leading zero components in each difference, thus achieving compression. For instance, the difference above $\langle 0, 0, 1, 01, 08\rangle$, is encoded into $\langle 2, 1, 01, 08\rangle$ since its first two components are zero. Fig. 4 shows the encoded blocks corresponding to Table (b).

3.3.7 Compression Efficiency

Let us define the efficiency μ of a compression method operating on a relation R by $\mu = 1 - C/D$, where D and C are the size of the relation before and after the differential coding step, respectively. Two factors affect the efficiency of TDC: compression overhead per tuple, and tuple spacings.

The compression overhead per tuple is the size of the count field used to indicate the number of leading zero components of a tuple difference. To avoid making the encoding scheme overly complex, we use a fixed-size field of size a bits to encode this count. Let R have n attribute domains. Since two distinct tuples cannot have a zero difference, the number of leading zero components in any difference tuple must be larger than zero but less than n. Thus, the number of bits required for the field is $a = \lceil \log_2 n \rceil$. If relation R has k tuples, the total compression overhead is $a(k - 1)$, since k tuples yield $k - 1$ differences.

The spacing between two tuples with respect to φ is measured by a function $\delta : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ defined as:

$$\delta(t_i, t_j) = \lceil \log_2(\varphi(t_j) - \varphi(t_i)) \rceil \tag{3.5}$$

for any two tuples $t_i \prec t_j$. The quantity $\delta(t_i, t_j)$ measures the number of bits required to represent the numerical difference between t_i and t_j. The further apart the tuples are, the larger is the difference, and thus the larger $\delta(t_i, t_j)$ is. Given a relation R, the mean spacing between tuples in R is:

$$\hat{\delta} = \frac{1}{k-1} \sum_{i=1}^{k-1} \delta(t_{i-1}, t_1) \tag{3.6}$$

for tuples $t_0, t_1, \ldots, t_{k-1}$. The mean spacing measures the average number of bits needed to encode a tuple difference.

Given a relation R, the total space requirements for the $k - 1$ tuple differences is $\sum_{i=1}^{k-1} \left(\delta\left(t_{i-1}, t_1\right) + \alpha \right)$ bits. Since the size of R before compression is $k \lceil \log_2 \varphi(t) \rceil$ bits where $t \in R$, the compression efficiency μ of TDC on R is given by

$A_{\tau(1)}$	$A_{\tau(2)}$	$A_{\tau(3)}$	$A_{\tau(4)}$	$A_{\tau(5)}$	$\mathcal{N}_R$
0	0	3	39	32	14816
0	1	0	40	40	18984
0	1	1	10	20	21140
0	2	1	38	35	39331
0	2	2	33	45	43117
0	2	3	34	20	47252
0	3	0	30	32	51104
1	0	0	49	30	68702
1	0	3	40	35	80419
1	1	0	50	20	85140
1	1	2	40	24	92696
1	2	0	41	22	100950
1	2	1	42	30	105118
1	2	2	56	25	110105
1	3	0	48	35	117795
1	3	2	38	40	125352
1	3	3	28	30	128798
2	0	0	50	30	134302
2	0	1	41	35	137827
2	1	0	38	32	149920
2	1	1	39	25	154073
2	1	2	40	25	158233
2	1	3	38	30	162206
2	2	2	27	43	173803
2	2	3	45	30	179038
2	3	0	40	20	182804
2	3	1	39	25	186841
2	3	2	40	20	190996
3	0	1	52	20	204052
3	0	2	47	20	207828
3	0	3	50	34	212130
3	1	0	60	35	216867
3	1	2	33	20	223316
3	1	3	34	28	227484
3	2	0	41	22	232022
3	2	1	29	35	235363
3	2	3	46	50	244658
3	3	0	41	30	248414
3	3	1	36	30	252190
3	3	2	23	25	255449

Table (a)

$A_{\tau(1)}$	$A_{\tau(2)}$	$A_{\tau(3)}$	$A_{\tau(4)}$	$A_{\tau(5)}$	R_d
0	0	3	39	32	14816
0	0	1	01	08	4168
0	0	0	33	44	2156
0	1	0	28	15	18191
0	2	2	33	45	43117
0	0	1	00	39	4135
0	0	0	60	12	3852
0	1	0	18	62	17598
1	0	3	40	35	80419
0	0	1	09	49	4721
0	0	1	54	04	7556
0	0	2	00	62	8254
1	2	1	42	30	105118
0	0	1	13	59	4987
0	0	1	56	10	7690
0	0	1	54	05	7557
1	3	3	28	30	128798
0	0	1	22	00	5504
0	0	0	55	05	3525
0	0	2	60	61	12093
2	1	1	39	25	154073
0	0	1	01	00	4160
0	0	0	62	05	3973
0	0	2	53	13	11597
2	2	3	45	30	179038
0	0	0	58	54	3766
0	0	0	63	05	4037
0	0	1	00	59	4155
3	0	1	52	20	204052
0	0	0	59	00	3776
0	0	1	03	14	4302
0	0	1	10	01	4737
3	1	2	33	20	223316
0	0	1	01	08	4168
0	0	1	06	58	4538
0	0	0	52	13	3341
3	2	3	46	50	244658
0	0	0	58	44	3756
0	0	0	59	00	3776
0	0	0	50	59	3259

Table (b)

Fig. 3. Tuple differential coding. Table (a) shows the tuples lexicographically reordered. Table (b) shows the tuples as differences. Column $\mathcal{N}_{\mathcal{R}}$ shows the result of mapping each tuple into a number by φ. Column R_d shows the result of taking the numerical difference between a number in $\mathcal{N}_{\mathcal{R}}$ and its preceding number.

$$\begin{aligned}\mu &= 1 - \frac{\sum_{i=1}^{k-1}\left(\delta\left(t_{i-1}, t_i\right) + \alpha\right)}{k\left\lceil \log_2 \varphi(t)\right\rceil} \\ &= 1 - \frac{(k-1)(\hat{\delta} + \alpha)}{k\left\lceil \log_2 \varphi(t)\right\rceil}\end{aligned} \tag{3.7}$$

where α is the fixed compression overhead per tuple.

An implication of the above is that positive efficiency is not guaranteed. Assuming that k is large, $k - 1 \approx k$, and $\mu = 1 - \left(\hat{\delta} + \alpha\right) / \left(\left\lceil \log_2\varphi(t)\right\rceil\right)$. If $\left\lceil \log_2\varphi(t)\right\rceil < \hat{\delta} + \alpha)$ then $\mu <$ 0. As the mean spacing, $\hat{\delta}$ is always less than $\lceil \log_2\varphi(t)\rceil$, α is the dominant factor that could make μ negative. A relation R is incompressible if the average difference between any two tuples is so large that the difference requires the same number of bits to store as the tuples. This happens when spacings between pairs of ordered tuples are very wide. However, this is pathological, and one is very likely to find clusters, for tuples tend to share attributes values, as in Table (a) in Fig. 2. In Section 5, we evaluate the efficiency of the technique both with simulated data as well as with real-world data in the form of the 1990 U.S. Census Public-Use Microdata Sample dataset [19]. All the results indicate high positive compression ratios.

3.4 Summary

In brief, we have discussed three database compression techniques. BIT and ATS are simple and intuitive applications of known techniques. TDC is specifically designed to compress tuples. How well they perform with respect to one another is studied in Section 5. The next section looks at issues related to the practical implementation of these techniques, in particular, at how they support standard database operations.

4 Tuple Access and Modification

Since our compression method is designed for use at the lowest levels of a database system, it is important to understand how it might interact with other system components, and particularly, whether its use might require changes to their structure. In this section, we demonstrate that no rethinking or redesign of other database system components is required, and that our method may be integrated cleanly with standard approaches to structuring them. In particular, we now consider how access mechanisms may be constructed on the coded tuples, and how the tuples may be retrieved and modified. Our focus is not on precise algorithms for these operations as but to give an idea of how our method may be integrated with standard access and retrieval mechanisms.

We have restricted our attention to these basic operations rather than to queries for several reasons:

1) All queries, simple or complex, reduce to a set of basic tuple operations.
2) The variety of queries is too large to derive a set of representative, typical queries.

The feasibility of these operations on a compressed database carries over to more complex queries which are built upon them. We have also used TDC as the compression technique in this section because it is a more complicated technique than BIT and ATS. The illustration extends to BIT and ATS compressed database.

4.1 Access Method

Fig. 4 shows an order-3 primary B^+ tree index constructed using the data blocks of Table (b) in Fig. 3. Notice that the search key in the index is an entire tuple. In conventional primary indices, the search key is usually a subset of attributes.

Operations on the tree-index are performed as usual. Suppose a query wishes to locate the tuple $\langle 2, 1, 3, 38, 30\rangle$ (which from Table (a) in Fig. 3 is located in block 6). 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, 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 based on the primary index above may be constructed. For ATS and BIT, the same mechanism can be used. Thus, we see that conventional access mechanisms are still applicable. Problems associated with these mechanisms such as the amount of tuples allocated per block and block overflows are similarly handled. An advantage of a compressed 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, we do not preclude the use of other methods such as hashing.

4.2 Tuple Insertion and Deletion

How are tuple insertion and deletion supported in a compressed database? Suppose we wish to insert in our previous database the tuple $t = \langle 1, 1, 0, 50, 21\rangle$. Using the primary index, we identify data block 3 as the block for insertion. The tuple is found to lie lexicographically between the second and third tuple in the block. Fig. 5 shows the result of tuple insertion. Notice that only tuple succeeding t is recoded, and that the changes are confined to the affected block. If the inserted tuple is lexicographically smaller than all the other tuples in the block, then it becomes the new reference tuple in the block. The process of tuple deletion is similar. Tuple modification is just a combination of tuple insertion and deletion.

4.3 Summary

In brief, we see that the integration of the three compression techniques with conventional access mechanisms satisfies the requirements for database compression described in Section 3. For TDC, all four requirements are met, although accessing a tuple within a block still requires the decompression of preceding tuples. Although ATS meets the first three requirements, it does not preserve tuple identity. BIT also meets all four requirements completely. Since bit-compressed tuples are fixed-size, a tuple can be accessed directly within a block by computing the correct offset from the beginning of the block. In Section 5, we undertake a more elaborate evaluation of these techniques.

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. We divide the evaluation into several parts. In Section 5.1, we look at compression ratios. In Section 5.2, we examine the time overhead of each of the compression techniques by measuring the average time to compress and decompress a disk block. In Section 5.3, we look at the effects of database compression on query response time. 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.

In addition to the performance evaluation on simulated data, we have also applied our compression technique to the compression of large real-world data sets in the form of the 1990 Public Use Microdata Samples (PUMS) from

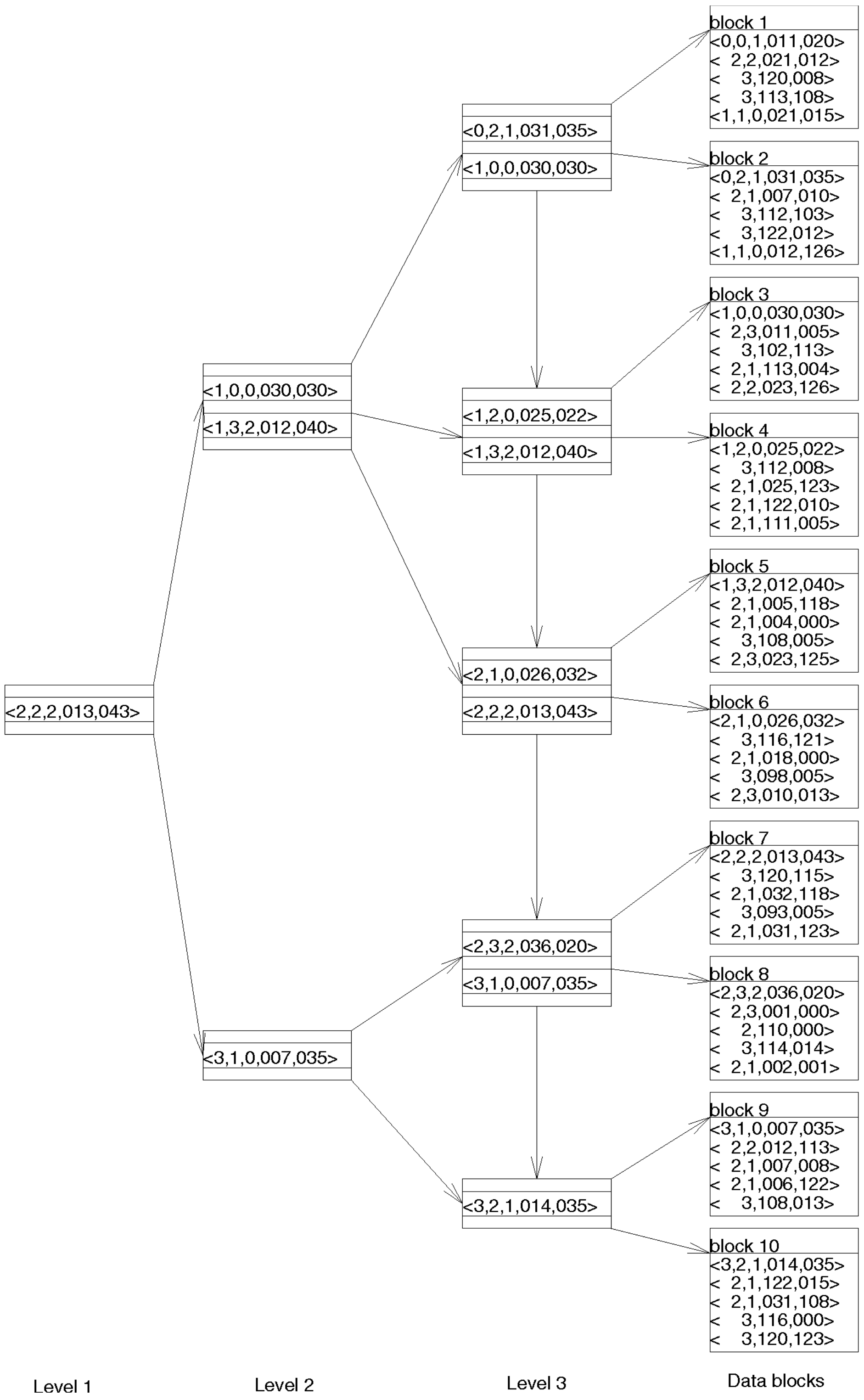


Fig. 4. Primary index. The data blocks contain difference tuples in φ order. Hence, the search key is an entire tuple. Each block begins with a *head* tuple. All tuples following the head tuple are difference tuples, in which the first integer is the count of leading zero components.

the U.S. Bureau of Census [19]. The results are reported in Section 5.4.

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),

BEFORE

$A_{\tau(1)}$	$A_{\tau(2)}$	$A_{\tau(3)}$	$A_{\tau(4)}$	$A_{\tau(5)}$
1	0	3	40	35
1	1	0	50	20
1	1	2	40	24
1	2	0	41	22

(a)

$A_{\tau(1)}$	$A_{\tau(2)}$	$A_{\tau(3)}$	$A_{\tau(4)}$	$A_{\tau(5)}$
1	0	3	40	35
0	0	1	09	49
0	0	1	54	04
0	0	2	00	62

(b)

AFTER

$A_{\tau(1)}$	$A_{\tau(2)}$	$A_{\tau(3)}$	$A_{\tau(4)}$	$A_{\tau(5)}$
1	0	3	40	35
1	1	0	50	20
1	1	0	50	21
1	1	2	40	24
1	2	0	41	22

(c)

$A_{\tau(1)}$	$A_{\tau(2)}$	$A_{\tau(3)}$	$A_{\tau(4)}$	$A_{\tau(5)}$
1	0	3	40	35
0	0	1	09	49
0	0	0	00	01
0	0	1	54	03
0	0	2	00	62

(d)

Fig. 5. Tuple insertion in block 3. Blocks (a) and (c) show the tuples before and after insertion; blocks (b) and (d) show their corresponding tuple differences.

2) variance in attribute domain size, and
3) attribute value skew.

The following variations are adopted throughout:

1) The domain size variance is low when the differences in domain sizes were no more than 10 percent of the average domain size. It is high when the differences were more than 100 percent.
2) The distribution of values within a domain is skewed when 60 percent of the values were drawn form 40 percent of the domain. When no skew existed, values were drawn uniformly from the domain.

The number of attribute domains of all relations was fixed at 8. We measured the number of disk blocks required by a relation under these variants.

Four sets of simulations were performed with these parameter variations. 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 sizes are varied in each of these combinations. These combinations are tabulated in Table (a) of Fig. 6. The results of the simulations are shown in Fig. 6. The following observations may be made:

1) The storage requirements are greatly reduced in a compressed relation. This is clear from the high positive compression efficiencies shown in Tables (b) through (d).
2) The compression figures in the TDC row are consistently higher than those in the BIT row, which are consistently higher than those in the ATS row. ATS, which uses the same technique as Unix's `compress` utility, displays its *typical* 50- to 60-percent range of compression ratios. BIT consistently reduces the database size to about a third of its original size. TDC outperforms the other two in all cases.
3) The efficiencies of both ATS and TDC improve with larger relations, as evidenced by the increasing ratios from Table (b) to (d). For TDC, larger relations have more number tuples and hence the mean spacing $\hat{\delta}$ (Equation 3.6) decreases. Therefore, the compression efficiency increases (Equation 3.7). TDC improves at a faster rate than ATS. BIT shows little or no improvement at all; thus, it is insensitive to the size of a relation.
4) For the same data skew, 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.
5) Data skew also affects compression efficiency. More data skew increases efficiency, as the figures in Tests 1 and 2 are relatively higher than the figures in Tests 3 and 4. A relation with a higher average data skew for each of its domain means that there is less randomness in the vslue distribution of each domain. This decreases $\hat{\delta}$, which increases μ, thus yielding better compressibility.

5.2 Compression and Decompression Time Overhead

We now measure the average time taken to compress a set of tuples whose compressed version can be allocated to a disk block with minimal unused space left in the block. We also measure the time to decompress the block.

The relation characteristics are as follows: We use a relation with 16 attributes of varying domain sizes. Each tuple is 35 bytes and there are 10^5 tuples in the relation. The block size is taken to be 8,192 bytes.

The measurements are made for each of the three techniques. For each of them, we perform the compression 100 times, and then the decompression 100 times. The average time for each operation is then computed. Before compression, 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 compression routine is the only user-level process executing in the system. They are taken from three different machines. The results are tabulated in Fig. 7 with the following observations:

1) The average time taken to compress and decompress a block is in the order BIT, TDC, and ATS.
2) Since compression/decompression are processor-bound, a faster processor yields lower time overhead. Thus, the time overheads improve with advances in processor technology.
3) ATS has the highest time overhead. BIT performs slightly better than TDC. However, it is to be noted

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

Table (a) Test characteristics

Technique	Test 1	Test 2	Test 3	Test 4
ATS	60.0%	52.4%	50.0%	47.4%
BIT	69.9%	67.7%	69.9%	67.7%
TDC	78.7%	74.4%	72.2%	68.8%

Table (b) 10^4 tuples

Technique	Test 1	Test 2	Test 3	Test 4
ATS	64.3%	56.5%	54.5%	50.0%
BIT	69.9%	67.7%	69.9%	67.7%
TDC	81.8%	77.8%	75.6%	72.2%

Table (c) 10^5 tuples

Technique	Test 1	Test 2	Test 3	Test 4
ATS	67.7%	56.5%	58.3%	52.3%
BIT	69.9%	67.7%	69.9%	67.7%
TDC	85.8%	80.8%	78.3%	75.0%

Table (d) 10^6 tuples

Fig. 6. Compression efficiency. The figures in the tables are the percentage reductions in relation size obtained via the formula: $(1 - a/b) \times 100\%$ where b and a are the size of the database before and after compression, respectively.

that although TDC does not perform a separate bit-compression step, it encodes the differences with as few bits as possible via run-length coding of leading zero components, so bit-compaction becomes an integral part of TDC. Thus, the actual computational overhead of tuple differencing is very little. For decompression on the HP 9000/735, the overhead is only 13.1 – 9.9 = 3.2 ms.

4) Combining the findings with that of the previous section, we may conclude that ATS is the worst in terms of compression efficiency and time. It is clearly inferior to TDC. Although BIT is slightly faster than TDC, its compression efficiency is worse than TDC and does not scale with relation size. Thus, TDC is the best overall algorithm so far.

5.3 Response Time

The previous two sections evaluated two inherent characteristics of compression techniques. What sets database compression techniques apart from data compression techniques in general is their influence on the processing of queries. We shall examine the effects of compression on the performance of queries in this section.

To calibrate our measurements, we need the notion of a typical query. This is difficult because there are many possibilities. Each query is specified by

1) the number of attributes involved,
2) the logical operators on these attributes, and
3) the arithmetic operations to be performed, etc.

To simplify things, we make the following assumptions:

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

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

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

$$C_1 = I + N(t_1 + t_2) \tag{5.1}$$

where I is the index search time, N is the number of disk blocks accessed, t_1 is the I/O time to read/write a block, and t_2 is the decompression time per block.

When the database is not compressed, the corresponding cost C_2, is

$$C_2 = I + N(t_1 + t_3) \tag{5.2}$$

where t_3 is the time to read and extract a block into a set of tuples. This time is included in t_2 where the result of decompression is a set of tuples. We shall now see how to estimate the various components.

5.3.1 Estimating t_1, t_2, t_3

The average I/O time per disk block, t_1, 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 over-

Technique	HP 9000/735	Sun 4/50	Dec 5000/120
ATS	20.5	25.8	72.3
BIT	10.4	11.1	29.7
TDC	13.7	25.8	50.4

Table (a) Average block compression time (milliseconds)

Technique	HP 9000/735	Sun 4/50	Dec 5000/120
ATS	21.3	26.9	64.1
BIT	9.9	10.0	26.0
TDC	13.1	13.1	34.8

Table (b) Average block decompression time, t_2 (milliseconds)

Fig. 7. Average compression/decompression time per 8 Kbytes block.

Attribute	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Uncompressed	167	167	167	167	167	167	167	167	167	167	165	158	143	136	98	1
ATS	85	84	84	84	85	85	84	84	85	85	84	83	84	82	73	1
BIT	59	59	59	59	59	59	59	59	59	59	59	59	59	59	57	1
TDC	21	25	35	45	53	53	53	53	53	53	53	53	53	53	52	1

Table (a) Number of blocks accessed for each attribute

Technique	Original number of blocks	Number of blocks accessed (N)	Savings ratio
Uncompressed	167	158.0	0.0%
ATS	85	83.4	47.2%
BIT	59	58.9	62.7%
TDC	53	47.2	70.1%

Table (b)

Fig. 8. Average number of blocks accessed. Column 2 in Table (b) gives the original number of blocks occupied by the relation. Column 3 is the average number of blocks accessed. Column 4 gives the proportion of block accessed saved. For instance, TDC reduces the number of blocks accessed by 70.1 percent.

head are usually in the range of 10–20 ms, 8 ms, and 2 ms, respectively [12]. Assuming a data transfer rate of 3 Mb/sec, the average I/O time for a block size of 8,192 bytes is $t_1 = 20\ ms + 8\ ms + (8{,}192\ b/3\ Mb)\ ms + 2\ ms \approx 30\ ms$. 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 sec-tion. The estimates for t_3 are: 1.34 ms for HP 9000/735, 2.92 ms for Sun 4/50, and 9.77 ms for DEC 5000/120.

5.3.2 Estimating N

We measure N via simulations. The selection query, as described above, has three *parameters*: *k, a, b*. Table (a) in Fig. 8 gives the number of blocks accessed when executing the query $\sigma_{A_k=a}(R)$ for each of the attributes of a tuple, i.e., $k = 1, 2, \ldots, 16$, and where $a = 0.5 \times |A_k|$. Observe that only one block is accessed for all techniques when $k = 16$ because A_{16} is the primary key. Table (b) shows the number of block accesses saved when the database is compressed.

5.3.3 Estimating I

The time (I) 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. For the reasons outlined below, we take it to be a constant component of both costs, independent of N. We have assumed A_k to be a nonprimary key attribute, so the index being searched is secondary. Secondary indices are generally smaller because the number of different attribute values of A_k is not as large as the number of primary key values. The actual number of values that appear in a relation is smaller still. Thus, the search time component of I is small and is dominated by the I/O needed to bring in the small number of index blocks. Here, we assume that the number of secondary index blocks is 5 percent of the total number of data blocks as shown in Fig. 9.

5.3.4 Results

Given the relation and query as described earlier, Fig. 10 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) by each of the three compression techniques, and when the relation is uncompressed (C_2). The following observations may be made:

1) Only BIT and TDC show positive improvements. ATS did not fare well because its decompression time overhead (t_2) is high. The other two techniques illustrate that query I/O time is reduced.
2) TDC outperforms BIT, mainly because it is more efficient than BIT in terms of compression. This helps to reduce TDC's N value.
3) The improvements are likely to improve in step with processor technology as the faster machines do better in Fig. 10. It is well known that processor technology progresses at a faster rate than disk technology. Thus, the t_2 component is likely to decrease, while t_1 stays about the same. TDC and BIT exhibit promise of improvement with processor technology.

A point to note about the performance figures is that the variable N in C_1, C_2 is a dominant factor in the amount of improvement. This value is shown in column 3 of Table (b) in Fig. 8, and is the average number of blocks accessed. The actual value of N depends on the attribute involved (k) and the values of the attributes (a, b). This may show variance, as evidenced by the TDC row in Table (a) of the same figure. The actual performance improvement depends on the set of frequently used attributes in the query selection criteria and their value distributions in the relation.

	Uncompressed	ATS	BIT	TDC
Number of data blocks	167	85	59	53
Number of index blocks, a	8.35	4.25	2.95	2.65
$I = a \times t_1$ (secs)	0.250	0.128	0.148	0.133

Fig. 9. Index search time *I*.

Technique	HP 9000/735	Sun 4/50	Dec 5000/120
Uncompresed, C_2	4.95	5.20	6.29
ATS, C_1	4.28	4.75	7.85
BIT, C_1	2.35	2.36	3.30
TDC, C_1	2.04	2.04	3.06

Table (a) Query I/O time (seconds)

Technique	HP 9000/735	Sun 4/50	Dec 5000/120
ATS	14.40%	8.65%	-24.80%
BIT	53.00%	54.62%	47.54%
TDC	59.20%	60.76%	51.40%

Table (a) Improvements

Fig. 10. Query I/O time and its improvements. The figures in Table (a) are computed using C_1 for ATS, BIT, and TDC, and C_2 for the uncompressed relation. The ratios in Table (b) are computed with the formula: $(1 - C_1/C_2) \times 100$ percent.

5.4 Experiments with Real-World Data

The previous sections examined the performance of the three compression techniques for simulated data. Although the size of the relation in terms of the number of tuples is large, the number of attributes of each relation is still comparatively small. In this section, we show how the performance comparison scales up to relations with large number of attributes using some very large real-world data.

The 1990 Public Use Microdata Samples (PUMS) from the U.S. Bureau of Census contain records representing 5-percent or 1-percent samples of the housing units surveyed in the U.S. and of persons residing in them. The 1-percent samples contain 2.3 million records and occupy 800 Mbytes, while the 5-percent samples contain 13 million records and occupy 4 gigabytes. Each record has approximately 150 attributes and occupies 232 bytes.

Census data is a particularly acute instance of the I/O bottleneck problem. The data, as seen here, is very large, and of indefinite retention. As new census data are collected, the amount of data in a collection will only increase. Adding to the problem is the fact that almost all queries to census data are aggregational. One usually must access the entire dataset in order to retrieve the relevant records for statistics computation. Thus, the queries are always I/O-intensive. There is also a need among demographers to *browse* through the census data on-line, in order to get a sense of the information contained in the data. Hence, real-time response is desired. Clearly, these are stringent requirements.

There are two relation schemas in the census dataset: *personal* and *household*. There are 124 attributes in the personal scheme. We projected 75 of these attributes into a subrelation which contains 472,980 records giving a total size of 83,244,480 bytes or approximately 80 megabytes. Thus, the subrelation is 10 percent the size of the 1-percent PUMS. We performed the same measurements as in the above sections. The results are tabulated in Fig. 11. We obtained the same results consistently on other subsets of the census data. We also obtained the same compression ratio when we compressed the entire census data set, although we performed no I/O time experiments on it. The measurements are performed on the HP 9000/735 machine only. The following observations may be made:

1) All techniques show positive compression and query time improvements, thus justifying the use of compression.
2) Due to higher number of attributes per tuple, the block decoding times of all techniques increase. In particular, the block decoding time for TDC shows the most significant increase. This forces the performance improvement ratios to drop.
3) TDC continues to outperform the other two techniques, although the differences are not as pronounced as with the simulated data.
4) ATS shows higher compression ratios than before because census data contains a lot of zeroes for many of the attribute values. This results in higher repetitions of zeroes, thus yielding higher compression ratios.

6 Related Work

Several techniques have been proposed for compressing statistical databases [1], [5], [7], [8], [9], [14], [21]. We will only discuss some of the more relevant work, particularly work on statistical databases with flat file structures, attribute transpositions and Huffman coding. The reader is referred to two survey papers on database compression by Bassiouni [2] and Severance [22] for more complete and detailed exposition.

	Uncompressed	ATS	BIT	TDC
Compression ratio	0.0%	76.2%	64.3%	75.6%
Block decoding time (msec)	5.0	43.7	21.4	46.3
Number of blocks accessed	898.8	277.0	387.5	242.8
Query I/O time (sec)	31.5	20.4	19.9	18.5
Performance improvements	0.0%	35.2%	36.8%	41.3%

Fig. 11. Performance improvements on census data.

```
H000000126010010030265000014020802000002004322000920000131112111421002550300011500020000000000···
P000000100100136202400000000224020000000013039000108999999100000002000040000000000001211023200···
P000000102100111402400000222230141000000013047000205999999100000002000000000000000000000000000···
H000000226010010030265000047011205000002006222050000000341113111722124000600036000500100003973···
P000000200000143000800000000101000000000431001000115050999100000002000020400100000042222001158···
P000000201100139000800000000231040000000431012000111032022100000002000040000000000002222036200···
P000000202100120400800000000106040000000431001000211050032100000002000040000000000002222011140···
P000000202100119400480000000236040000000431012000111050032100000002000040000000000002222011125···
P000000211000122400480000000106000000000187001000110939999201001002000040000000000002222003200···
```

Fig. 12. Census records.

6.1 Constants Removal

In [8], [9], [14], the authors are concerned with statistical databases that assume a flat file structure, i.e., those consisting of one or more sequential file(s) of bytes. Such databases usually contain numeric data, say from the results of laboratory experiments, monitoring of seismic activities, or business trends (see characteristics 3 and 4 in Section 1). Such databases exhibit little or no record structure. Consequently, the preservation of structure in a compressed database is not important. In contrast, the structure of many important classes of statistical databases (the U.S. Census data, for example) is very record-oriented.

A subset of the 1990 U.S. census records is shown in Fig. 12. There are nine records shown (two begin with H, and the others with P), all concatenated together without any structure. Notice that the attribute values are discrete and there are many runs of zeroes.

Indeed, the primary concern of such techniques is the removal of *constants* from the databases. Constants are runs of identical data values, such as zeroes, that are usually removed or coded using run-length coding or its variants. As the database is a contiguous sequence of bytes, much of the work is concerned with the determination of efficient *mappings* between the uncompressed database (a flat file) and the compressed database (a file containing uncoded bytes and run-length codings).

Some of the issues addressed by these methods are:

1) the maintenance of mappings when new records are inserted as more data are gathered,
2) the provision of run-length codings of different constant types, and
3) the provision of efficient and random access to the encoded file.

BIT and TDC are different from these techniques in that the record structures are preserved. This allows the database to be treated like a relational database, thus harnessing known and standard access methods and operations for record manipulation. We are now working on extending the compression techniques to relational databases in general.

6.2 Attribute Transposition

Attribute transposition [4], [27] stores a relation as a collection of contiguous attribute columns, where all values for an attribute domain are stored together. Work in this area has concentrated on different schemes for encoding attributes columns. Since attribute values are repeated, the standard coding methods may be used to replace them with smaller sized codewords to achieve compression [5].

Bit-level compression is identical in concept to attribute-level compression except that it is carried to the extreme [29]. Each column of attribute values is further vertically partitioned into single-bit columns, each corresponding to the binary pattern of each attribute value.

Both forms of transposition have their strengths and weaknesses with respect to tuple-wise encoding (BIT and TDC) or block-wise encoding (ATS). They permit the selective retrieval of columns required for query processing. They are comparatively faster if the desired set of columns in the query is very much smaller than the entire set of attributes. However, since numerous smaller column files may be generated by transposition, there is a higher level of disk block fragmentation. This may result in higher seek times when locating attributes. In addition, it may no longer be economical to construct indices for each of the columns for random access. Thus, decoding may have to be performed serially.

6.3 Huffman Coding

Huffman coding is popular technique for database compression at the character level [3], [7]. Each character is replaced by a codeword whose size is inversely proportional to the frequency of occurrence of that character. As we have discussed in Section 2, Huffman coding is a coding method based on statistical modeling, and faces complications when new data are inserted or deleted. In these cases, the old data have to be recoded. In addition, performing compression at the character-level is usually time-consuming, as the ATS technique has illustrated.

7 Conclusions

We have shown in this paper how database compression is different from data compression in general. On the basis of this difference and the characteristics of statistical databases, we have designed and tested three database compression techniques. These techniques are suitable for databases because they are able to support standard database processing while the database is in a compressed state.

Of the three techniques, Tuple Differential Coding (TDC) has been shown to have the best performance. TDC is effectively a combination of three basic compression techniques: textual substitution, bit-compaction and differential coding. While the first two are generally applicable to most other data, the form of differential coding incorporated by TDC is adapted to a table of tuples. The result is a compression technique customized for relational database compression. Because the tuple structures are still intact, TDC is able to support on-line normal database and querying operations while the database is still compressed.

ACKNOWLEDGMENT

This work was supported in part by the National Aeronautics and Space Administration through the Consortium for International Earth Science Information Networking.

REFERENCES

[1] P. Alsberg, "Space and Time Savings through Large Database Compression and Dynamic Restructuring," *Proc. IEEE*, vol. 63, pp. 1,114–1,122, Aug. 1975.

[2] M.A. Bassiouni, "Data Compression in Scientific and Statistical Databases," *IEEE Trans. Software Eng.*, vol. 11, no. 10, pp. 1,047-1,058, Oct. 1985.

[3] M.A. Bassiouni and K. Hazboun, "Utilization of Character Reference Locality for Efficient Storage of Databases," *Proc. Second Int'l Workshop Statistical Database Management*, pp. 338-344, Sept. 1983.

[4] D.S. Batory, "On Searching Transposed Files," *ACM Trans. Database Systems*, vol. 4, no. 4, pp. 531-544, Dec. 1979.

[5] D.S. Batory, "Index Coding: A Compression Technique for Large Statistical Databases," *Proc. Second Int'l Workshop Statistical Database Management*, pp. 306–314, Sept. 1983.

[6] T.C. Bell, J.G. Cleary, and I.H. Witten, *Text Compression*. Englewood Cliffs, N.J.: Prentice Hall, 1990.

[7] G.V. Cormack, "Data Compression on a Database System," *Comm. ACM*, vol. 28, no.12, pp. 1,336-1,342, Dec. 1985.

[8] S.J. Eggers and A. Shoshani, "Efficient Access of Compressed Data. *Proc. Sixth Int'l Conf. Very Large Databases*, pp. 205–211, 1980.

[9] S.J. Eggers, F. Olken, and A. Shoshani, "A Compression Technique for Large Statistical Databases," *Proc. Seventh Int'l Conf. Very Large Data Bases*, pp. 424–434, 1981.

[10] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W.H. Freeman, 1979.

[11] S.W. Golomb, "Run-Length Encodings," *IEEE Trans. Information Theory*, vol. 12, pp. 399-401, July 1966.

[12] R.H. Katz, G.A. Gibson, and D.A. Patterson, "Disk System Architectures for High Performance Computing," *Proc. IEEE*, vol. 77, no. 12, pp. 1,842-1,858, Dec. 1989.

[13] G.G. Langdon, "An Introduction to Arithmetic Coding," *IBM J. Research and Development*, vol. 28, no. 2, pp. 135-149, 1984.

[14] J.Z. Li, D. Rotem, and H.K.T. Wong, "A New Compression Method with Fast Searching on Large Databases," *Proc. 13th Int'l Conf. Very Large Databases*, pp. 311–318, 1987.

[15] W.K. Ng and C.V. Ravishankar, "Attribute Enumerative Coding: A Compression Technique for Tuple Data Structures," *Proc. Fourth Data Compression Conf.*, p. 461, Snowbird, Utah, Mar. 29–31, 1994.

[16] W.K. Ng and C.V. Ravishankar, "Data Compression System and Method Representing Records as Differences between Sorted Domain Ordinals Representing Field Values," U.S. Patent No. 5,603,022, Feb. 1997.

[17] W.K. Ng and C.V. Ravishankar, "A Physical Storage Model for Efficient Statistical Query Processing," *Proc. Seventh IEEE Int'l Working Conf. Statistical and Scientific Databases*, pp. 97–106, Charlottesville, Va., Sept. 28–30, 1994.

[18] W.K. Ng and C.V. Ravishankar, "Relational Database Compression Using Augmented Vector Quantization," *Proc. 11th IEEE Int'l Conf. Data Eng.*, pp. 540–549, Taipei, Taiwan, Mar. 6–10, 1995.

[19] "Census of Population and Housing, 1990: Public Use Microdata Samples U.S.," machine readable data files prepared by the Bureau of the Census. Washington, D.C., 1992.

[20] J.J. Rissanen and G.G. Langdon, "Universal Modeling and Coding," *IEEE Trans. Information Theory*, vol. 27, no. 1, pp. 12-23, 1981.

[21] F. Rubin, "Experiments in Text File Compression," *Comm. ACM*, vol. 19, no. 11, pp. 617-623, Nov. 1976.

[22] D.G. Severance, "A Practitioner's Guide to Database Compression: Tutorial," *Information Systems*, vol. 8, no. 1, pp. 51-62, 1983.

[23] C.E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical J.*, vol. 27, no. 3, pp.379-423, 1948.

[24] A. Shoshani and H.K.T. Wong, "Statistical and Scientific Database Issues," *IEEE Trans. Software Eng.*, vol. 11, no. 10, pp. 1,040-1,047, Oct. 1985.

[25] H. Tanaka and A. Leon-Garcia, "Efficient Run-Length Encodings," *IEEE Trans. Information Theory*, vol. 28, no. 6, pp. 880-890, June 1982.

[26] T.A. Welch, "A Technique for High Performance Data Compression," *Computer*, vol. 17, no. 6, pp. 8-19, June 1984.

[27] G. Wiederhold, *Database Design*, second edition. New York: McGraw-Hill, 1983.

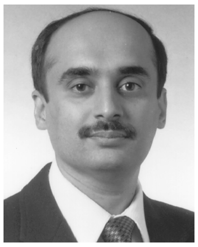
[28] R.N. Williams, *Adaptive Data Compression*. Boston: Kluwer Academic, 1991.

[29] H.K.T. Wong, H.F Liu, F. Olken, D. Rotem, and L. Wong, "Bit Transposed Files," *Proc. 11th Int'l Conf. Very Large Databases*, pp. 448-457, 1985.

[30] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Trans. Information Theory*, vol. 23, no. 3, pp. 337-343, 1977.

[31] J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Trans. Information Theory*, vol. 24, no. 5, pp. 530-536, 1978.

Wee Keong Ng received his BS degree in computer science with honors from the National University of Singapore in 1990, and his MS and PhD degrees in computer sciences from the University of Michigan, Ann Arbor, in 1992 and 1996, respectively. He has been with the School of Applied Science at the Nanyang Technological University, Singaore, since 1996. His research interests include statistical databases, data/database compression, evolutionary algorithms, digital libraries, and World Wide Web databases. Dr. Ng is a member of the IEEE, the IEEE Computer Society, and the ACM.



Chinya V. Ravishankar received his BTech degree in chemical engineering from the Indian Institute of Technology, Bombay, in 1975; and his MS and PhD degrees in computer sciences from the University of Wisconsin–Madison in 1986 and 1987, respectively. He has been with the Electrical Engineering and Computer Science Department at the University of Michigan, Ann Arbor, since 1986. His teaching and research at the University of Michigan has been in the areas of databases, distributed systems, and programming languages. Dr. Ravishankar is a member of the Software Systems Research Laboratory, which he founded, and of the Real-Time Computing Laboratory at the University of Michigan. His present research interests include data bases and large-scale distributed systems. He is a member of the IEEE Computer Society and the ACM.