Database Compression: A Performance Enhancement Tool

Gautam Ray [*] Jayant R. Haritsa [*] S. Seshadri [†]

[*] Dept. of Computer Science and Automation
Indian Institute of Science, Bangalore 560012, India
{gautam, haritsa}@csa.iisc.ernet.in

[†] Dept. of Computer Science and Engineering
Indian Institute of Technology, Bombay 400076, India
seshadri@cse.iitb.ernet.in

Abstract

Compression is typically used for databases that have grown large enough to create a strain on system storage capacity. We argue here that database compression is attractive from a query processing viewpoint also and should therefore be implemented even when disk storage is plentiful. We study the compression ratio and query processing performance of a variety of compression algorithms, for different compression granularities, on a set of relations drawn from real world databases. Our study shows that attribute level compression is the best from a query processing perspective but has poor compression ratio. We then present a modified attribute level compression algorithm, based on non-adaptive arithmetic compression, called **COLA**, *which simultaneously provides good query processing and reasonable compression ratios. We also analyze, for a range of relational queries, the performance benefits that* COLA *could be expected to provide.*

1 Introduction

Many database management systems provide support, to a limited extent, for users to store data in *compressed* form. The main motivation for providing compression, in both the commercial products and in the research literature, has been the savings in disk storage. Compression is therefore recommended to be used for databases that have grown so large as to create a strain on system storage capacity.

In this paper, we argue that database compression is attractive not only from a storage reduction viewpoint, but also from a performance viewpoint. In fact, we claim that database compression should *always* be implemented, *even* when disk storage is plentiful. This is because, as shown in this paper, a properly designed compression scheme can provide significant improvement in **query processing** performance (measured in disk accesses). Since query processing is the essential function of database systems, compression should primarily be viewed as an effective performance enhancement tool.

To derive improved query processing performance, the database compression scheme has to be designed carefully. To motivate this, consider a typical SQL query such as

Select * from STUDENT where Roll_Number between 5000 and 6000 and City = "Bombay"

which retrieves, from the STUDENT relation, all those students from Bombay city whose roll number is between 5000 and 6000. Assume that the STUDENT relation is 2000 pages long, that it is ordered and (B-tree) indexed on the roll number attribute. Also assume that roll numbers range from 1 to 10000, which means that the tuples with roll numbers between 5000 and 6000 are located in pages 1000 through 1200 of the relation (assuming a packed file).

In the absence of compression, it is easy to deduce that answering the above query will initially take h disk accesses to access the first relevant page through the index (where h is the height of the index), and then an additional 200 disk accesses to retrieve the data pages and perform the selection based on city. Since h is typically only 3 or 4, the query will be answered in less than 205 disk accesses.

Now, consider the case where the file corresponding to the STUDENT relation has been compressed, say by half, using a high-quality compression algorithm such as compress in Unix or pkzip in DOS (compression factors of 50 percent are quite common with these algorithms). Assuming a uniform degree of compression, we can expect that the tuples with roll numbers between 5000 and 6000 will now be in pages 500 through 600 of the compressed file. Again, as in the uncompressed case, the first relevant page can be accessed through the index and then 100 disk accesses are required to retrieve the compressed data pages. However, in order to *decode* these 100 pages, *all* the pages from the beginning of the compressed relation until page 500 also have to be decompressed! This is because compression techniques such as compress incrementally build decoding tables during the decompression phase based on the data encountered so far (in the earlier pages). In other words, their *context* of compression covers the entire file. So, the query processing cost in the compressed case (equal to $h + 600$) is almost three times the cost in the uncompressed case! In addition, *every* tuple in the (100) relevant data pages has to be decompressed to check whether the student is from Bombay city. This will involve significant decompression overhead at the CPU.

Examples of the above type have led database users to employ compression only as a last resort when storage is scarce. However, the problem lies more in the *implementation* of compression rather than in compression itself. To understand this, consider an alternative scheme where compression is done at the *attribute* level, that is, where each attribute of each record is compressed separately, resulting in the compression context being localized to the attribute. In this case, the above query may be answered quite efficiently: Just as in the file compression case, the 100 relevant pages are accessed through the index. In contrast, however, the initial 500 pages do not have to be decompressed because of the localized context. The selection predicate is then implemented as follows: Compute the compressed equivalent of the attribute value "Bombay", and for each student tuple, compare its compressed city attribute value with this value. Whenever a match is found, the corresponding tuple is decompressed and returned to the user.

In the above attribute compression scheme, query processing requires only about half the number of disk accesses ($h + 100$) as compared to the uncompressed case (and $1/6$ as compared to the file compression case). Another important feature of the scheme is that the significant decompression overhead that was incurred in the file compression case is kept to the minimum possible here since *only* the result tuples, that is tuples useful to the user, are decompressed. Viewed differently, the query has been executed completely in the *compressed domain itself*. We will use the term "precise decompression" to refer to processing strategies where decompression is limited to only the result tuples.

A word of caution, however, with regard to the above numbers: Our claim that attribute compression requires only $1/6$ the number of disk accesses as compared to the file compression case implicitly assumes that the compression ratios for both file compression and attribute compression are comparable. However, as later, this may typically not be the case since the reduced context creates lesser compression opportunity, resulting in lower compression ratios for attribute level compression.

Goals

From the above discussion, we see that our goal is to develop a database compression scheme that simultaneously provides efficient query processing, precise decompression and good compression ratio. In designing such a scheme, there are three main questions that need to be answered:

- What compression algorithm should be used? There are several well-known compression algorithms such as Huffman coding, Arithmetic coding, LZW algorithm, etc. Further, as explained later, there are both adaptive and non-adaptive flavors of some of these algorithms.

- What compression granularity should be used? As discussed above, file level compression is clearly untenable for query processing. The other possible compression granularities are page, record, and attribute. However, in going from file compression to other levels of compression, there may be a fall in compression ratio.

- Should meta-data also be compressed? Since meta-data in the form of catalogs or indexes are small in comparison to data files, it appears that the small storage savings that are realized by compression are more than offset by the increased processing complexity. A related question is whether the indexes should be built using the original keys or their compressed equivalents. Here again, it appears that retaining the original keys is far more attractive since the lexical *ordering* provided by the index leaf level is retained.

In this paper, we study the compression ratio performance of a variety of compression algorithms for different compression granularities. The experiments were conducted on a set of relations drawn from real-world databases including a bibliographic database and a bank database. Our experiments show that non-adaptive algorithms perform better than their adaptive counterparts and that Arithmetic coding has the best overall performance. We also argue that attribute level compression is the best from a query processing perspective. However, our experiments show that its compression ratio can be considerably worse than that of the other compression granularities. To address this issue, we present a modified attribute level compression algorithm based on non-adaptive Arithmetic coding called **COLA**, which simultaneously provides good query processing and reasonable compression ratios. Finally, we analyze, for a range of relational queries, the performance benefits that COLA could be expected to provide.

Related Work

The general ideas of database compression have been discussed in [RH93, Sev83, Cor85, Bas85, Reg81]. These studies evaluate the various types of compression techniques from a storage perspective but do not consider their suitability for database query processing. Considerable work has been done in the field of compression of scientific and statistical databases (e.g. [Bat83, EOS81, McC82, BH83, LST83]). However, the nature of data and the types of operations in these databases differ significantly from those in commercial databases, which is our target domain. In the last few years, a couple of studies [Gra93, BW94] have considered issues similar to those considered in this paper. While [Gra93] brings out the potential of compression with respect to query processing, [BW94] is an excellent investigation of the implementation practicality of these ideas. Our work builds on their efforts and attempts to analyze these issues in more detail, especially from the query processing perspective.

2 Compression: Pros and Cons

In this section, we describe the benefits and drawbacks typically associated with compression. For the drawbacks, we discuss how these may be addressed in modern database systems.

Apart from the query processing improvement mentioned in the Introduction, compression results in several other benefits also [GS91, Bas85, BW94, Gra93] by virtue of storing data in a reduced space: Firstly, disk seek times are reduced since the compressed data fits into a smaller physical disk area. Secondly, related objects can be clustered closer together. Thirdly, data compression increases disk bandwidth by increasing the *information density* of the transferred data. Finally, network communication costs are reduced in distributed databases and client-server applications due to reduced data transfer.

From a transaction processing perspective, there are two further benefits [Gra93]: First, the buffer hit rate increases since a larger fraction of the database now fits into the buffer pool. Second, the disk I/O to log devices is decreased since the log records are shorter.

The drawbacks typically associated with compression [Bas85, Gra93, BW94] are: Firstly, data compression and decompression may result in considerable overhead at the CPU. This overhead can be minimized if the compression scheme ensures *precise* decompression, as explained in the Introduction. Secondly, compression results in data records becoming variable-sized. However, many modern database systems are already equipped to handle variable-sized records in order to efficiently support schema transformation [BW94] or variable length attributes (as required in SQL92), so this is not a real problem. Thirdly, compression schemes pose a problem with regard to precisely (in the decompression sense) processing comparative predicates such as "less than" or "greater than" since the lexical relationships between data values may not be retained by their compressed equivalents. It may be possible to address this issue by using recently proposed order preserving compression algorithms [ZIL93], if they turn out to be efficient. Finally, compression, by virtue of reducing redundancy in data, reduces the ability to recover from errors. For example, a single bit error in the output may result in the decoder misinterpreting all subsequent bits. Problems of this nature, however, are taken care of by current communication protocols and disk controllers.

3 Compression Techniques

Most data compression techniques are based on one of two models: **statistical** or **dictionary**. In statistical modeling, each distinct *character* of the input data is encoded, with the code assignment being based on the probability of the character's appearance in the data. In contrast, dictionary-based compression schemes maintain a *dictionary* which contains a list of commonly occurring character strings in the data and their corresponding codes. While encoding, these schemes search for the longest *string* of input characters that also appear in their dictionary. Once this string match is identified, the code of the matched string is used in place of the entire character string.

Yet another dimension of lossless compression algorithms is that they may be **adaptive** or **non-adaptive**. In adaptive schemes no prior knowledge about the input data is assumed and statistics are dynamically gathered and updated during the encoding phase itself. On the other hand, non-adaptive schemes are essentially "two-pass" over the input data: during the first pass, statistics are gathered, and in the second pass, these values are used for encoding.

In this study, we have considered three popular compression techniques: **Huffman**, **Arithmetic** and **LZW**. The Huffman coding and Arithmetic coding techniques implement the statistical model, while the LZW scheme is dictionary-based. For Huffman and Arithmetic, we have experimented with both adaptive and non-adaptive flavors, but for LZW, we have used only its adaptive version since the non-adaptive version is comparatively complex to implement. In addition to these techniques, we have also evaluated the simple Run-length encoding (RLE) scheme which is supported in many current database systems (e.g. IMS [BW94]). The RLE scheme does not use either the statistical or the dictionary model – it simply recognizes successive repetitions of characters.

In the remainder of this section, we describe the salient features of the above-mentioned compression algorithms.

3.1 Huffman Coding

In Huffman coding [Huf52], a tree is constructed with the characters of the input alphabet forming the leaves of the tree. The links in the tree are labeled with either 0 or 1 and the code for a character is the label sequence that is obtained by traversing the path from the root to the leaf node corresponding to that character in the Huffman tree. The tree is built such that the most frequent characters in the input data are assigned shorter codes and the less frequent characters are assigned longer codes.

As mentioned earlier, both adaptive and non-adaptive versions of Huffman coding exist. In non-adaptive Huffman coding, the Huffman tree is completely built before encoding starts, using the known frequency distribution of the characters in the data to be compressed. The tree remains unchanged for the entire duration of the encoding process. The decoder builds the same tree using the same frequency distribution before decoding the compressed data. On the other hand, adaptive Huffman coding starts off with a Huffman tree that is built using an *assumed* frequency distribution of the characters in the input data. A common practice is to assume that all characters are equally likely to occur. As the encoding process proceeds and more data is scanned, the Huffman tree is modified based on the data seen up to that point. Therefore, the Huffman tree changes dynamically during the encoding phase and the same character can have different codes depending on its position in the data being compressed (unlike non-adaptive Huffman).

3.2 Arithmetic Coding

In Huffman coding, each character is encoded into an *integral* number of bits. This means that the codes may often be longer than that strictly required for the character. For example, a character with probability of occurrence 0.9 can be coded minimally in 0.135 bits (from information-theoretic considerations), but requires 1 full bit in this scheme.

Arithmetic coding attempts to address the above shortcoming of Huffman coding. Here, the compressed version of the input data is represented by the interval between two real numbers of *arbitrary* precision, (x, y), where $0 \leq x < y \leq 1$. At the start, the range is initialized to the entire interval [0,1), and this range is progressively refined. During the encoding process, each character is assigned an interval within the current range, the width of the interval being proportional to the probability of occurrence of that character. The range is then narrowed to that portion of the current range which is allocated to this character. So, as encoding proceeds and more data is scanned, the interval needed to represent the data becomes smaller and smaller, and the number of bits needed to specify the interval grows. The more likely characters reduce the range less than the unlikely characters and hence add fewer bits to the compressed data. The implementation details of this scheme are given in [Wit87, Jon88].

Arithmetic coding also has adaptive and non-adaptive versions, in exactly the same manner as that described previously for Huffman coding.

3.3 LZW Coding

The LZW (Lempel-Ziv-Welch) algorithm [Wel84] is a popular compression technique, used in both Unix compress and DOS pkzip. The scheme is organized around a *string translation table*. This table contains a set of character strings and their corresponding code values. The string table has a prefix property that for every string in the table its prefix is also in the table. That is, if string ωK, composed of some string ω and some single character K, is in the table, then ω is also in the table.

In LZW, the input data is scanned sequentially and the longest *recognized* input string (that is, a string which already exists in the string table) is parsed off each time. The recognized string is then replaced by its associated code. Each parsed input string, when extended by its next input character, gives a string that is not yet present in the string table. This new string is added to the string table and is assigned a unique code value. In this manner, the string table is built incrementally during the compression process.

3.4 Runlength Coding

Runlength coding (RLE) is an extremely simple and old compression technique. It takes advantage of consecutive repetitions (or *runs*) of the same character. For example, consider the string "cccccaabbbb". In normal 8-bit ASCII representation, the string would require 11 bytes (since the string has 11 characters). RLE, however, can encode the string in 8 bytes, as "ϕc5aaϕb4". In this coding scheme, ϕ is a special character (usually a non-printable character such as ASCII 255), which denotes the beginning of a run. It is followed by the repeated character and the length of its run.

4 Compression Granularity

In a typical relational database system, compression can be conceptually applied at four different levels, namely the *file level*, the *page level*, the *record level* and the *attribute level*. At the file level, each relation in the database is compressed as a whole. Since compression techniques generally work better with larger data sets, we may expect that the best compression ratios would be realized at this level. In particular, adaptive models may find the large size favorable (since they have to dynamically gather statistics). With respect to query processing, however, file level compression requires the entire relation to be decompressed each time data in the relation is accessed (as illustrated in the example in the Introduction). Further, if the relation is modified (insert, delete or update), the whole relation has to be recompressed. This compression/decompression overhead may result in extremely slow query processing times.

An alternative scheme is for relations to be compressed at the page level. Since the page size in most commercial databases is around 4K, this gives scope for deriving good compression ratios. However, page compression has a problem similar to that described above for file compression: The whole page has to be decompressed and possibly re-compressed when data within the page is referenced or modified. This may again result in poor query processing times. Moreover, since the compressed pages are of variable size, additional complications arise: Firstly, a compressed page will occupy only a fraction of a disk block (assuming that the uncompressed pages are of disk block size, as is usually the case). Since disk transfer is usually in units of a disk block, fetching the compressed page will also bring in unnecessary data corresponding to other pages. Secondly, compressed pages may cross disk block boundaries. So, two disk block accesses have to be made to fetch the single compressed page to memory. Thirdly, when data in a page is updated, the size of the compressed page may change. In that case, the page has to be relocated to some other block, creating a hole in its previous position.

The next level of compression is the record level. Since, on average, record sizes vary between 40 to 120 bytes [BW94], the limited data size may cause a significant fall in the compression ratio, especially for adaptive algorithms. However, from the query processing viewpoint, this option is more attractive since only the records that *potentially* contribute to the result need to be decompressed and compressed. The decompression overhead is also limited due to the small size of a record. Since record level compression deals with fixed size pages, we can use disk-block sized pages to facilitate efficient disk I/O. Moreover, with fixed sized pages, the buffer manager of the DBMS does not have to be modified to account for compression. The only additional memory requirement is a record sized buffer which is used to hold uncompressed records when needed.

The lowest possible level at which compression can be done is at the attribute level. The common data types of attributes are integers, floating point numbers and character strings. Generally, the size of integers varies from 1 to 4 bytes, that of floating point numbers from 4 to 8 bytes and that of characters strings from 10 to 32 bytes [BW94]. This means that the data size is really limited and may result in poor compression ratios, especially for adaptive techniques. From the query processing viewpoint, however, the attribute level appears to be the most attractive because it permits **precise** queries, that is, where decompression is necessary only for the result tuples, as illustrated in the example in the Introduction. This ability to execute a query entirely in the compressed domain appears extremely desirable from a performance viewpoint. Further, even if the need arises to decompress/compress an attribute, only that attribute and not the entire tuple needs to be decompressed/compressed. This is a distinct advantage over record level compression where entire tuples need to be decompressed/compressed. In addition, attribute level compression can be implemented with fixed sized pages, as in the case of record level compression.

Apart from having different compression ratios on the input data, each of the above granularities has different amounts of space overhead involved in their implementations. These overheads reduce the *effective* compression ratio. In certain situations, as explained in Section 6, the overheads may even exceed the compression ratio, leading to an *expansion* of the input file! Therefore, it is critical to include overhead effects in evaluating compression schemes, and this aspect is analyzed below.

4.1 Overheads

In file level compression, the file is compressed as a whole and no overhead is involved. In page level compression, pointers are stored for random access to the variable sized pages. The number of such pointers is equal to the number of pages in the relation, and therefore the space overhead as a fraction of relation size is insignificant. In record level compression, due to the variable-sized records, a pointer to the beginning of each record has to be maintained. So the percentage overhead is directly proportional to the compression ratio. At the attribute level, a pointer is stored for each attribute of each record. This overhead can, therefore, be significant, especially if the number of attributes in the relation is large. We will hereafter use the term **pointer overhead** to refer to the storage overhead arising out of pointers. An important point to note here is that the problem of tracking variable sized attributes is *inherent* to databases that support variable sized attributes (which many modern databases do). So, the pointer overhead that arises out of compression cannot really be considered as an additional overhead in these systems since the same overheads will also be present in the uncompressed case. However, in order to be *conservative* in our estimates of performance improvement due to compression, we assume that pointer overhead is present *only* for the compressed files.

Apart from pointer overhead, there is another type of overhead which may become significant only in the case of attribute level compression: Each attribute when compressed may not occupy an integral number of bytes in general. But the compressed attributes are stored in an integral number of bytes so that tracking and accessing the attributes does not become cumbersome. So in general the last few bits in the last byte of a compressed attribute is wasted as no useful information is stored in them. On average, a half-byte may be wasted for each attribute, and this may become significant if the relation has many small-sized attributes. We will hereafter use the term **padding overhead** to refer to this overhead.

4.2 Summary

From the above discussions, we observe that there are inherent difficulties in simultaneously achieving the desired goals of efficient query processing, precise queries, and good compression ratio. To quantitatively evaluate the performance effects and the various tradeoffs involved, we conducted a set of experiments which are discussed in the following sections.

5 Experiments

The goal of our experiments was to evaluate the compression ratio performance as a function of compression algorithm and compression granularity. Three relations drawn from real world databases were used as inputs to the experiments: a relation from a bibliography database, a relation from a bank database, and a synthetically generated relation that was constructed based on value distributions observed in real data. In this section, we describe the compression granularities, the compression schemes and the input relations that were considered in our experiments.

5.1 Compression Granularities

Compression performance was evaluated at all four granularities, namely, file, page, record and attribute levels. In file level compression, the entire source relation was compressed as a whole. In page level compression, each individual page of the source relation was compressed separately. On compression, therefore, each page became variable sized. The individual pages were tracked to provide for random access. In record level compression, the compressed file consists of fixed size pages. Each record after compression was placed contiguously in a page until the page became full. Then the next page was filled, and so on. In attribute level compression too, the compressed relation consists of a number of fixed sized pages. The individual attributes of a record were compressed, concatenated to form a record, and stored contiguously in a page until the page became full. Then the next page was filled, and so on.

In all the above cases, the additional pointers required to locate data (as discussed in Section 4) were stored. For record and attribute level compression, the compressed records were filled into pages such that they did not cross page boundaries.

5.2 Compression Schemes

The following lossless compression algorithms were implemented: Run-length coding, LZW coding, adaptive and non-adaptive Huffman coding, adaptive and non-adaptive Arithmetic coding. Each of the algorithms was implemented at all four compression granularities (file, page, record and attribute). To verify that the encoding algorithms worked correctly, the corresponding decoding algorithms were also implemented at all the four levels.

For the non-adaptive schemes, the entire relation was scanned once to gather relevant statistics. Based on these statistics, the relations were compressed. In the adaptive schemes, the algorithms gathered statistics and simultaneously compressed the data based on the statistics gathered up to that point. An important point to note here is the *scope* of the statistics used to compress data at any point of time. In non-adaptive case, the entire relation is scanned first. So, the scope of the statistics, used to compress data at any point in time, for any of the four levels of compression, is the entire relation. This is not the case with adaptive schemes where the statistics used to compress data at any point in time depends on the level at which compression is being done. For file level compression, the scope of the statistics used refers to the entire file scanned up to the point. On the other hand, for page level compression, the scope of the statistics used refers to the portion of the page scanned up to that point. Similarly, the scope of the statistics gathered is limited to individual records and attributes in the case of record level and attribute level compression, respectively.

5.3 Relations

Having described the compression schemes, we now move on to discussing the relations on which these compression schemes were operated. As mentioned earlier, the relation suite includes a bibliographic relation, a bank relation and a synthetic relation. The bibliographic relation and the bank relation were obtained from a public archive and from a major bank, respectively.

The bibliographic relation contains names of authors, titles, publishers, etc. of various scientific publications. Each tuple in the relation has 11 attributes which are all character strings and the size of each tuple is 1828 bytes. The number of records is 80690.

Table 1 : Statistics of Relations							
Relation Name	Number of Attributes	Character Strings	Numeric	Record Size	Number of Records	Records per Page	Unused Space per Page
PUB	11	11	0	1828	80690	2	428
BANK	24	19	5	236	55862	17	72
NUM	12	1	11	76	9964	53	56

The bank relation contains information about customer accounts. Each tuple consists of 24 attributes, of which 19 are character strings and 5 are numeric. The size of each tuple is 236 bytes and the relation has 55862 records.

The attributes of the above two relations are predominantly character strings. In order to also include numeric data, a third file was constructed synthetically consisting of records having mostly numeric attributes (the attribute values were derived from distributions that were fitted to real data). Each tuple of this relation has 12 attributes, and of these only one is a character string with the remaining 11 being numeric attributes. The size of each tuple is 76 bytes and the relation has 9964 records.

In the remainder of this paper we will refer to the bibliographic relation, the bank relation and the synthetic numeric relation as **PUB**, **BANK**, and **NUM** respectively. The above relation statistics are summarized in Table 1. The table also describes the number of records per page, the unusable space at the end of each page (since records do not cross page boundaries) and the total number of pages. These statistics are computed assuming the following physical implementation of relations: Each relation is a file consisting of 4K pages and each page has a 12 byte header information. The rest of the page (4084 bytes) is used to store the data records.

6 Results

Having described the experimental framework, we move on to presenting and analyzing the results of our experiments in this section. The performance measure in these experiments is the percentage compression, PC, which is defined as

$$PC = (1 - \frac{size\ of\ compressed\ data}{size\ of\ uncompressed\ data}) * 100$$

that is, it is the percentage equivalent of the compression ratio. In general, PC is a number that ranges between 0 and 100 (higher values correspond to greater degrees of compression). In some of our experiments, however, there was an *expansion* in the data instead of the intended compression – in such cases, PC has a negative value.

To help explain the observed PC results, our programs kept track of the *average*, *maximum* and *minimum sizes* of the compressed records, as well as the *variance* in the size of the compressed records. We also monitored the *average overhead*, which is the average number of bytes wasted per page to store pointers for tracking variable sized records/attributes[1], and the *average fragmentation*, which is the average amount of unused space left per page on average due to not allowing records to cross page boundaries. (Note that these statistics are relevant only for record and attribute level compression).

Table 2: Percentage Compression For PUB

Method	**File**	**Page**	**Record**	**Attribute**
RLE	80.35	80.00	79.1	77.50
LZW	91.57	82.90	79.18	70.23
AA	79.45	76.10	72.40	60.00
AH	75.58	73.95	71.80	66.50
NAA	78.82	78.70	78.01	76.57
NAH	74.91	74.86	74.40	72.20

Table 3: Percentage Compression For BANK

Method	**File**	**Page**	**Record**	**Attribute**
RLE	24.40	24.20	22.00	11.00
LZW	81.00	65.00	36.78	-19.56
AA	51.36	48.43	29.45	-6.27
AH	49.79	47.50	35.44	-3.13
NAA	46.41	46.35	44.87	26.50
NAH	43.45	43.40	41.20	28.47

Table 4: Percentage Compression For NUM

Method	**File**	**Page**	**Record**	**Attribute**
RLE	21.54	21.0	18.60	4.25
LZW	46.20	40.59	20.63	-33.33
AA	44.19	42.48	20.74	-21.20
AH	44.02	38.67	13.83	-2.66
NAA	44.38	44.33	40.21	19.57
NAH	43.85	43.80	39.89	20.74

The percentage compression achieved for the relations **PUB**, **BANK** and **NUM** are shown in Tables 2, 3 and 4, respectively. In these tables, the percentage compression is listed for each compression algorithm and for each compression granularity. The following acronyms have been used to identify the compression algorithms: **RLE** for Run-length

[1] The padding overhead was also measured and found to be insignificant compared to the pointer overhead.

coding, **LZW** for Lempel-Ziv coding, **AA** for adaptive Arithmetic coding, **NAA** for non-adaptive Arithmetic coding, **AH** for adaptive Huffman coding and **NAH** for non-adaptive Huffman coding.

In Tables 2, 3 and 4, we first note that the general compression levels are highest for PUB (around 75 percent) and comparatively smaller for BANK and NUM (less than 50 percent). This can be attributed to the fact that PUB is purely a textual database while BANK and NUM have significant numeric attributes which are less amenable to compression. Observe that even in NUM, which is almost exclusively numeric, compression levels of upto 40 percent are achievable for certain algorithm-granularity combinations (NAA or NAH with Record level).

Another point to note is that the non-adaptive algorithms almost always outperform their adaptive counterparts. This is especially so with regard to attribute level compression, where the adaptive algorithms result in expansion, instead of compression (for NUM, LZW has an expansion of 33 percent!).

Finally, we observe that the compression levels generally decrease from file level compression to attribute level compression for all of the considered algorithms.

We now discuss the performance of each compression algorithm in detail. The analysis concentrates on record and attribute levels because, as we have already observed, file or page levels are unsuitable for efficient query processing.

RLE : The percentage compression for RLE is unexpectedly good for PUB, whereas for BANK and NUM, it is significantly worse than some of the more sophisticated techniques. The good compression ratio for PUB is due to the fact that in PUB, the average sizes of most of the attributes is much smaller than their maximum allocated size (this fact was experimentally confirmed). As a result, long runs of blanks are there in most fields – a perfect opportunity for RLE. In fact, this type of skewed data brings out the inadequacy of using fixed length records in databases and highlights the need for supporting variable sized records.

The deficiencies of RLE become apparent when the percentage compressions for BANK and NUM are studied. It is also observed that the compression ratio declines from file level to attribute level. This is because of the increasing pointer overhead with finer granularity. (In RLE, there is no padding overhead because records or attributes are always encoded in integral number of bytes.) In summary, it appears that although RLE is simple to implement and fast to execute, its effectiveness and use appears limited to primarily those databases that are almost entirely comprised of textual data.

Adaptive Schemes : It is seen that for adaptive schemes the compression ratio falls significantly as the unit of compression changes decreases. The reduction as we move from the page level to the record level is due to two reasons. Firstly, the scope to adapt is further reduced in this case and is confined to one record length. Secondly, the pointer overhead comes into play. Finally, as we move to the attribute level, we find that the relations are expanded, rather than compressed. This is because, when the scope to adapt is confined to the length of a single attribute, the compression levels are rather low, and the effect of the pointer overhead predominates with the net

Table 5 :	Percentage Compression
Relation	**COLA Coding**
PUB	79.54
BANK	44.21
NUM	21.27

result that the compression achieved is offset and the relations expand. The LZW is the worst affected in this respect, followed by adaptive Arithmetic and then adaptive Huffman. In summary, it appears that although adaptive techniques may be suitable for file or page level compression, they are not suitable for record level compression and infeasible for attribute level.

Non-adaptive Schemes : The non-adaptive Arithmetic and Huffman schemes produce comparable results in all cases at all levels. It is observed that the compression achieved at the record is slightly less than that at the page level due to pointer overhead. However, the reduction in compression from the record to attribute level is more significant. This is mainly because of the increased pointer overhead and to some extent due to the padding overhead.

6.1 Column-Based Attribute Compression

From the above analysis, it appears that attribute level compression results in comparatively poor compression ratios. However, from the query processing perspective, the ideal compression level is attribute, as discussed earlier. To address this dilemma, we investigated the possibility of devising alternative attribute compression schemes that would provide reasonably good compression ratio. In particular, we tried the following scheme: Instead of having a single frequency distribution for the entire relation, use a separate frequency distribution table for *each* attribute in a relation. The motivation for this approach is that data belonging to the same attribute is usually semantically related and is expected to have similar distribution. Therefore, the characteristics of each attribute are reflected more accurately and the smoothing out of the peculiarities of a particular attribute (which may happen in the case of a single relation-wide frequency distribution) is prevented.

The above column-based approach was used in combination with non-adaptive Arithmetic coding to design an attribute level compression scheme, which we call **COLA** (Column-Based Attribute Level Non-Adaptive Arithmetic Coding). The COLA algorithm was tested on the three relations used in the earlier set of experiments. The results obtained from this experiment are listed in Table 5.

In this table, we see that the compression percentages show a significant improvement over the attribute level compression with a single frequency table, except in the case of NUM where the improvement is only marginal. For PUB, the compression achieved now is better than even the corresponding file level compression (Table 2)! For BANK it is virtually the same as that achieved at the record level. In NUM, however, this scheme fails to match the record level compression. This is because almost all the attributes are numbers which are similarly distributed and no additional semantics is captured by maintaining frequency tables for each column. However, in general, different columns in a relation may be expected to have unique characteristics of their own – in such cases, maintaining multiple frequency tables may be of significant benefit as observed for PUB and BANK.

Since attribute level encoding is best suited for query processing and since COLA is expected to give results that are, on an average, comparable with record level compression, we can conclude that COLA [2] appears to be a viable approach to database compression. In the next section, we study the effectiveness of COLA with respect to query processing for a variety of relational queries.

7 Query Processing

In this section, we analyze the query processing performance that the COLA database compression scheme (Section 6) could be expected to provide for relational operators such as *select*, *project*, *join* and *updates*. Since join is the most complex relational operator, we use two examples of equijoin to highlight the performance effects. The first is a nested-loop block join, and the second is a join with a clustering index on the join attribute of one of the join relations. Apart from join, we have also analyzed the performance improvement for a number of other queries – these results are available in [Ray95].

In the analysis, we use the following notation: T_R denotes the number of tuples in relation R, B_R denotes the number of blocks in which these tuples fit when stored packed (contiguously), and $I_{R.A}$ is the expected number of different values of attribute A that are found in relation R. Further, the compressed version of relation R is denoted by R_c, and the number of blocks required to store this compressed relation is denoted by B_{R_c}.

The following assumptions are made to facilitate our analysis: (1) We assume that the join attributes are **compression dependent**. Two attributes are said to be compression dependent if they are compressed using the same statistics. Since it is usually meaningful to perform a join only on those attributes that have the same data type and are semantically related, using the same statistics to compress them is not expected to result in degraded compression ratios. The important point to note here is that an equi-join on compression dependent attributes can be *precisely* decompressed. (2) A uniform distribution of values is assumed for all attributes. Therefore, the expected number of tuples having a particular value of attribute A in relation R is given by $T_R/I_{R.A}$. (3) The cost of accessing the index is ignored. This is because the cost of index access is the same for both the uncompressed and the compressed cases since the indices are kept uncompressed (as mentioned in Section 1). In addition, the contribution of indexing to the total cost in terms of disk accesses is marginal. (4) The page size is equal to the size of a disk block. This implies that fetching a page from disk involves one disk access. (5) The total cost of a relational operation is expressed in terms of the number of disk block accesses required for that operation. Although, compression/decompression overhead is a CPU cost, we *normalize* it to

[2]It may appear feasible to have a COLA algorithm wherein the column statistics are obtained *adaptively*. We have not considered this variant since query processing would then require, for the decompression of an attribute, decompression of all the preceding values in the column.

an equivalent number of disk accesses to facilitate comparison. For normalizing, the time required for decompression of relevant data is divided by the time to access one disk block, denoted by t_{da}. (6) The cost of decompressing a byte of compressed data is given by t_{db} This cost, expressed in terms of number of disk accesses, becomes (t_{db}/t_{da}).

The following parameter values are assumed: (1) The time required for a disk access, t_{da}, is 10ms [CL88]. (2) The time required to decompress a byte, t_{db}, is 1 μsec. This is deliberately chosen to be on the higher side (decompressing a byte typically takes 15 machine instructions [BW94], and most current processors, including PCs, have more than 15MIPS rating) in order to ensure that the claimed performance improvements due to compression are conservative. (3) The compression factor (PC) is 50 percent (this figure is based on the experimental evidence presented in Section 6).

With the above assumptions, we now move on to computing the cost of joining two representative relations $R(A, B)$ and $S(B, C)$ on attribute B. The statistics of relations R and S are assumed to be as follows:

- For relation R, $T_R = 100000$, $B_R = 10000$, $B_{R_c} = 5000$, $I_{R.B} = 10000$.
- For relation S, $T_S = 100000$, $B_S = 5000$, $B_{S_c} = 2500$, $I_{S.B} = 10000$.

7.1 Nested-loop Block Join

We first compute the cost of joining R and S using a nested-loop block join algorithm [KS91]. Assuming that M memory buffers are available, the join procedure is as follows:
1. Divide S into segments of $M - 1$ blocks each.
2. Read the first segment of S and in the remaining buffer cycle through all the blocks of R. Produce the join of each tuple in the segment with each tuple in the block that match on the join attribute.
3. Repeat Step 2 for each segment of S.

During each invocation of Step 3 in the above algorithm, each block of S is read once, and each block of R is read once for each segment in S. As the number of segments in S is approximately $\lceil B_{S_c}/(M-1) \rceil$, the input cost of the compressed join (measured in disk accesses) is

$$B_{R_c} \lceil B_{S_c}/(M-1) \rceil + B_{S_c}$$

Including the output cost of the equijoin, the total cost is

$$B_{R_c} \lceil B_{S_c}/(M-1) \rceil + B_{S_c} + (B_{R_c} T_S + B_{S_c} T_R)/max(I_{R.B}, I_{S.B}) + DCost \quad (1)$$

where $DCost$ is the decompression cost of the result tuples that are presented to the user.

The expected number of tuples in the output of the join is $T_R T_S / max(I_{R.B}, I_{S.B})$. Further, it has been observed in [BW94] that tuples are typically 40 to 120 bytes long. This means that, assuming 50% compression, the sizes of the compressed tuples range between 20 to 60 bytes. For this example, we make the very conservative assumption that the average size of a compressed tuple is 100 bytes. With these assumptions, $DCost$ evaluates to

$$DCost = T_R T_S / max(I_{R.B}, I_{S.B}) * 100 * (t_{db}/t_{da}) = 10000 \ disk \ accesses$$

for the example relations R and S. It should be noted that since the join attributes are assumed to be compression dependent, no decompression is required for matching join attribute values.

Using the above decompression cost and substituting the parameters of the example relations R and S in Equation 1, we compute the overall cost of the compressed join, for a buffer pool of size $M = 101$, to be 212500 disk accesses.

The corresponding cost for the *uncompressed* join is given by

$$B_R \lceil B_S/(M-1) \rceil + B_S + (B_R T_S + B_S T_R)/max(I_{R.B}, I_{S.B}) \tag{2}$$

and substituting the parameter values yields a cost of 655000 disk accesses, which implies that the compressed join is almost 70 percent cheaper than the uncompressed join.

7.2 Clustered Index Join

We now compute the cost of joining R and S using a clustered index join algorithm [KS91]. Assuming that relation S has a clustering index on B, the join procedure is as follows:
1. Read the first block of R.
2. For each tuple of the block, use the index to find the matching tuples of S. Since the index is not stored compressed, the attribute B has to be decompressed for each tuple of R in order to retrieve the corresponding tuples of S via the index.
3. Repeat the above steps for all the blocks of R.

For each tuple in R, $1/I_{S.B}$ of the total tuples of S have to be fetched which implies that $max(1, B_{S_c}/I_{S.B})$ block accesses have to be made. This results in a total of $T_R\ max(1, B_{S_c}/I_{S.B})$ block accesses. In addition, each block of R has to be read once. Finally, we have to include the cost of writing out the result relation and the decompression cost involved in the entire operation. Therefore, the total cost comes to

$$B_{R_c} + T_R\ max(1, B_{S_c}/I_{S.B}) + (B_{R_c} T_S + T_R B_{S_c})/max(I_{R.B}, I_{S.B}) + DCost \tag{3}$$

The decompression cost, $DCost$, is the sum of the decompression cost of the join attribute of R and the decompression cost of the result tuples that are presented to the user. The decompression cost of the join attribute depends on the size of the compressed attribute. Integer or floating point attributes are usually 4 bytes long and character strings are typically 10 to 40 bytes long in their uncompressed form [BW94]. Assuming 50% compression, the maximum size of the compressed join attribute will therefore normally not exceed 20 bytes. Accordingly, the maximum cost of decompressing the join attribute in all the tuples of R is

$$T_R * 20 * t_{db}/t_{da} = 200\ disk\ accesses$$

The decompression cost of the result tuples is 10000 disk accesses, as shown earlier in the nested join example.

Using the above decompression costs and substituting the parameters of the example relations R and S in Equation 3, the overall cost of the compressed join evaluates to 190200 disk accesses.

The corresponding cost for the *uncompressed* join is given by

$$B_R + T_R\ max(1, B_S/I_{S.B}) + (B_R T_S + T_R B_S)/max(I_{R.B}, I_{S.B}) \tag{4}$$

and substituting the parameter values yields a cost of 260000 disk accesses, which implies that the compressed join is almost 30 percent cheaper than the uncompressed join.

Summary

The above examples show that significant query processing improvement can be derived through the COLA scheme for equi-join operations. In a similar manner, the reduction in cost for other relational operators can also be derived – a detailed discussion is available in [Ray95].

8 Conclusions

Data compression has traditionally been used primarily for reducing disk space usage in database management systems. In this paper, we have attempted to demonstrate that data compression could lead to significant savings during query processing. Before incorporating compression in a database system, two fundamental choices about the compression algorithm and the compression granularity have to be exercised. We evaluated the compression ratio performance of Huffman coding, Arithmetic coding, LZW coding and RLE coding, for file, page, record and attribute granularities, on both text-based and numeric-based relations. Our study revealed that attribute level compression is highly desirable from a query processing perspective but yields a poor compression ratio. We addressed this problem by modifying attribute level compression to operate on a column basis with regard to obtaining frequency distribution statistics. The motivation for this approach was that values in a column are typically semantically related and therefore offer greater scope for compression. The column-based approach was combined with non-adaptive Arithmetic compression, which had the best overall performance in our compression ratio experiments, to create the COLA algorithm. This algorithm simultaneously provides good query processing and reasonable compression ratios. A detailed analysis of a variety of relational queries shows that COLA could be expected to provide significant improvements in practice.

This results of this study demonstrate that data compression can be a useful performance enhancement tool which not only reduces disk space usage but also provides significant performance savings during query processing. We plan to further strengthen this result by implementing COLA in a database management system and timing the performance of various queries from a standard benchmark. Also, while COLA works well for text-based relations, its compression ratio performance for numeric attributes is not as good. We are currently working on modifying COLA to rectify this shortcoming. Another direction of study is to evaluate the compression efficacy of order preserving compression algorithms [ZIL93]. This would permit use of compressed values in an ordered index and also allow perform operations such as sorting to be executed entirely in the compressed domain.

Acknowledgements

The work of J. R. Haritsa was supported in part by a research grant from the Dept. of Science and Technology, Govt. of India.
We thank Joy Thomas and S. Sudarshan for their valuable assistance in providing us with reference material.

References

[Bas85] M. A. Bassiouni. "Data Compression in Scientific and Statistical Databases," *IEEE Trans. on Software Eng.*, October 1985.

[Bat83] D. Batory. "Index Coding: A Compression Technique for Large Statistical Databases," *Proc. of 2nd Intl. Workshop on Statistical Database Management*, September 1983.

[BH83] M. Bassiouni and K. Hazboun. "Utilization of Character Reference Locality for Efficient Storage of Databases," *Proc. of 2nd Intl. Workshop on Statistical Database Management*, September 1983.

[BW94] B. R. Iyer and D. Wilhite. "Data Compression Support in Databases," *Proc. of VLDB Conf.*, September 1994.

[CL88] M. J. Carey and M. Livny. "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication," *Proc. of VLDB Conf.*, August 1988.

[Cor85] G. V. Cormack. "Data Compression in Database Systems," *Comm. of ACM*, December 1985.

[EOS81] S. Eggers, F. Olken, and A. Shoshani. "A Compression Technique for Large Statistical databases," *Proc. of VLDB Conf.*, September 1981.

[Gra93] G. Graefe. "Options in Physical Database," *SIGMOD Record*, September 1993.

[GS91] G. Graefe and L. Shapiro. "Data Compression and Database Performance," *Proc. of ACM/IEEE Computer Science Symp. on Applied Computing*, April 1991.

[Huf52] D. A. Huffman. "A Method for Construction of Minimum-Redundancy Codes," *Proc. of the IRE*, September 1952.

[Jon88] D. W. Jones. "Application of Splay Trees to Data Compression," *Comm. of ACM*, August 1988.

[KS91] H. Korth, and A. Silberschatz. *Database System Concepts*, 2nd ed., McGraw-Hill, 1991.

[LST83] E. Lefons, A. Silvestri, and F. Tangorra. "An Analytic Approach to Statistical Databases," *Proc. of VLDB Conf.*, October 1983.

[McC82] J. McCarthy. "Metadata Management for Large Statistical Databases," *Proc. of VLDB Conf.*, September 1982.

[Ray95] G. Ray. "Data Compression in Databases," *Master's Thesis*, Dept. of Computer Science and Automation, Indian Institute of Science, June 1995.

[Reg81] H. K. Reghbati. "An Overview of Compression Techniques," *IEEE Computer*, April 1981.

[RH93] M. A. Roth and S. J. Van Horn. "Database Compression," *SIGMOD Record*, September 1993.

[Sev83] G. D. Severance. "A Practitioner's Guide to Database Compression – A Tutorial," *Information Systems*, January 1983.

[Wel84] T. A. Welch. "A Technique for High Performance Data Compression," *IEEE Computer*, June 1984.

[Wit87] I. H. Witten, et al. "Arithmetic Coding For Data Compression," *Comm. of ACM*, June 1987.

[ZIL93] A. Zandi, B. Iyer, and G. Langdon. "Sort Order Preserving Data Compression for Extended Alphabets," *Data Compression Conf.*, 1993.