DATA COMPRESSION IN DATABASES

A Project Report

Submitted in partial fulfilment of the

requirements for the Degree of

Master of Engineering

in

COMPUTER SCIENCE & ENGINEERING

by

GAUTAM RAY

Department of Computer Science and Automation

Indian Institute of Science

Bangalore – 560 012

June 1995

Contents

Acknowledgements **iv**

Abstract **v**

1 Introduction **1**
1.1 Database Compression 1
1.2 The Benefits 6
1.3 The Drawbacks 7
1.4 Conclusion 8

2 Compression Techniques **10**
2.1 Introduction 10
2.2 Compression Models 11
2.3 Runlength Coding 11
2.4 LZW Coding 12
2.5 Huffman Coding 14
2.5.1 Non-adaptive Huffman Coding 14
2.5.2 Adaptive Huffman Coding 17
2.6 Arithmetic Coding 20
2.6.1 Non-adaptive Arithmetic Coding 25
2.6.2 Adaptive Arithmetic Coding 25
2.7 Conclusion 25

3 Design Issues **26**
3.1 Introduction 26
3.2 Hardware Versus Software Compression 26
3.3 Adaptive Versus Non-adaptive Model 27
3.4 Unit of Compression 28
3.5 Query Types 30
3.6 Conclusion 30

4 Implementation **31**
4.1 Introduction 31
4.2 The Relations 32
4.3 Compression Scheme 32

5 Results and Analysis **35**
5.1 Introduction . . . 35
5.2 Results . . . 35
5.3 Analysis . . . 36

6 Query Processing **46**
6.1 Introduction . . . 46
6.2 Selection . . . 50
6.2.1 Exact Match Selection . . . 50
6.2.2 Range Selection . . . 53
6.3 Project . . . 55
6.4 Join . . . 56
6.4.1 No Index on Join Attributes Exists . . . 57
6.4.2 Index on Join Attribute Exists . . . 61
6.5 Update Operators . . . 64
6.5.1 Insert . . . 65
6.5.2 Delete . . . 66
6.5.3 Update . . . 68
6.6 Conclusion . . . 70

A Output Cost of Joins **72**

Bibliography **73**

Acknowledgements

I am extremely grateful to my guide, **Dr. Jayant Haritsa**, for introducing me to an interesting research topic and for providing me with constant guidance and encouragement throughout the duration of my project. He has always helped me with his valuable suggestions; even with the writing of this report.

I am also thankful to all my instructors, who have contributed immensely to my knowledge and self-improvement. My special note of thanks to all the personnel at SERC for their help and co-operation throughout the course.

I would also like to take this opportunity to thank all my fellow students for the wonderful time we had together. Particularly, I am grateful to Sanjib (Sanjib Kr. Ghosh, CSA,ME(I)) for his co-operation and delightful company throughout the four years.

Finally, I am deeply indebted to my parents who have always encouraged me through the good and not-so-good times. Last, but not the least, I extend my gratitude to all my friends back home, who have always encouraged me in all my endeavors. Specially, I would like to thank Nabaneeta, who has been a constant source of inspiration for me.

Abstract

Data compression is considered to be an useful tool for saving disk storage for quite sometimes now. But the need and usefulness of data compression in databases has not been addressed very seriously till now. Usual performance enhancement techniques like sophisticated disk placement and disk scheduling policies, intelligent clustering schemes, complex query optimization methods, appropriate indexing and hashing are widely used in DBMS to improve performance which manifests itself in terms of reduced average response time of transactions. Data compression, when used to supplement these conventional techniques, can further reduce the response time of transactions, primarily by reducing the number of disk accesses while processing queries. In this thesis, we have explored the various issues and tradeoffs involved in database compression. We have also suggested an **attribute-level** compression scheme that performs well for all the sample relations we considered and is expected to do so for relations from a wide variety of relational databases. The impact of such an attribute level compression scheme on query processing performance has also been studied in detail.

Chapter 1

Introduction

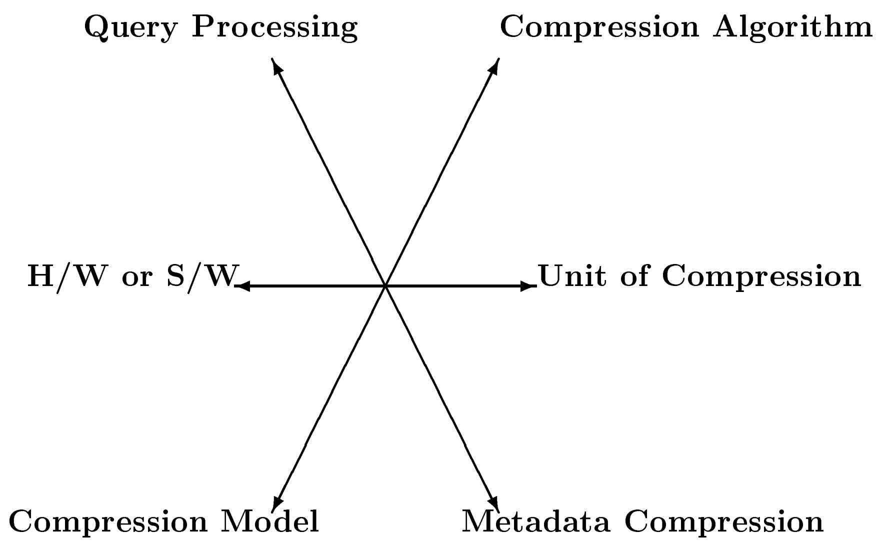
1.1 Database Compression

In Information Theory the term **entropy** is used to measure the amount of information contained in a data set. The higher the entropy of a data set, the more information it contains. The entropy of an entire data set is simply the sum of the entropy of all the individual symbols in the data set. The entropy of a symbol is defined as the negative logarithm of its probability of occurrence. To determine the information content of a symbol in bits, its entropy is expressed using base 2 logarithm. For example, if the probability of occurrence of character 'e' in a piece of English text is 1/16, the information content of the character is 4 bits. So the character string "eeeee" has a total information content of 20 bits. If standard 8 bit ASCII is used to encode this string, we are using 40 bits. The difference between the 20 bits of entropy and the 40 bits used to encode the data is where the potential for **data compression** arises. The goal of data compression is to reduce this difference as far as possible.

The need and usefulness of data compression for storing large text files, source codes, executables etc. is well recognized. Most popular operating systems like UNIX, and DOS come equipped with software support for file compression. For example, UNIX *compress/uncompress*, DOS *pkzip/pkunzip* are being widely used for quite sometimes now. However, **database compression** is a field which is still relatively unexplored. Data compression, when applied to data in a database, is termed as database compression. In recent years the need to store and process large volumes of data has increased significantly. This has led to a sharp rise in the number of large database installations. The principal bottleneck for these databases are the **disk storage costs** and **performance penalties** during query processing. Advances in technology has reduced the cost of disk storage drastically over the past few years. But the increased need to store voluminous data has outpaced the decrease in storage costs. So disk storage still remains an important cost factor, particularly for large databases. If data in a database is stored compressed, significant savings in disk storage can be expected. Another major concern in large databases is the degradation in query processing performance. As the size of the database increases, more and more data has to be scanned to answer queries. Inevitably, the response time of transactions increases. Sophisticated disk placement and disk scheduling policies, complex query optimizing techniques and suitable indexing and hashing schemes have to be used to keep the response time of transactions within tolerable limits. There has been a lot of research in these fields. Surprisingly, the potential of data compression as a tool to optimize database performance has not been considered very seriously. Database compression can, in fact, produce significant improvement in query processing, primarily by reducing the number of disk accesses. The general ideas of database compression has been discussed in [RH93, Sev83, Cor85, Bas85, Reg81]. These studies mostly explore the various types of compression techniques but do not consider their suitability for database compression. Moreover, only very few studies [Grf93, BW94] have considered the design issues and tradeoffs involved in order to integrate data compression with a DBMS or the improvements in query processing as a result of database compression. In this thesis, we have tried to make a detailed study of how to provide compression support within the framework of a relational DBMS. In the process, we have considered the various design issues involved and the resultant performance improvement achieved.

The primary objective in file compression is saving storage. Large text files, source codes or executable files are compressed and stored on disk to save disk space. All that is important here, is a good **compression ratio**, CR, defined as

Figure 1.1: Issues in Database Compression



CR = (1 - size of compressed data/size of uncompressed data) The compression and decompression overhead is not very critical because these files are compressed *as a whole* before storage and decompressed *as a whole* when needed. In databases too, a good compression ratio is useful to reduce disk space requirement. But there are some other constraints as well, which gives rise to additional performance metrics like **compression/decompression overhead** and **number of disk block accesses** during query processing. In databases, frequent operations are performed on data. If data is kept compressed and data has to be decompressed for comparison, the CPU overhead may become intolerable. So it is desirable to implement database compression in such a manner so as to minimize the compression/decompression required during query processing. Further, disk accesses is by far the costliest operation during query processing. So, the impact of database compression on disk accesses has to to carefully considered. Keeping in view the three performance metrics, namely, compression ratio, compression/decompression overhead and number of disk accesses, the goal of the project can be summed up as: *to achieve a significant compression ratio, and at the same time do as little compression/decompression and disk I/O as possible while processing queries.* In order to achieve this objective, a host of design issues must be considered, namely the **choice between hardware and software compression**, the **unit of compression**, the **compression algorithms** and **compression models** that can be used, the impact of compression on **different query types** and whether to use **metadata compression**. The issues are summarized in Figure 1.1.

Hardware compression is fast but has to be provided by the vendor. Software compression is slow and is only possible if the compression/decompression overhead is kept low. In a relational database system, a database consists of a number of relations. Each relation relation is usually stored in a separate file. Such a file consists of a number of fixed sized pages. Each page has a page header and a number of fixed sized records. The header keeps track of book-keeping information like the number of records in a page, the vacant slots in a page etc. Each record, in its turn, consists of a number of attributes, each attribute having a fixed size. This structure of a relational database indicates that compression can be applied to databases at four different levels, namely the **file**, the **page**, the **record** or the **attribute level**. We have dealt with all the levels of compression in this study and considered the merits and drawbacks of each of them. As one moves from file level to attribute level, some additional overheads appear. These overheads are of two types: **pointer overhead** and **padding overhead**. The pointer overhead arises because of the need to keep track of variable sized records/attributes. The padding overhead appears because of the need to *pad* a compressed unit of data (the unit can be the entire file, a page, a record or an attribute) to the next nearest byte boundary. This is done for faster and easier access of data. Another important aspect of database compression is the choice of the compression algorithm. If the algorithm is too complex the compression/decompression overhead may be too high. On the other hand, if the algorithm is very simple, the compression may not be adequate. We have considered some of the most widely used techniques, namely, RLE, LZW, Huffman coding and Arithmetic coding. Moreover, one has to consider the compression model to be used as well. There are two types of compression model: **adaptive** and **non-adaptive**. The adaptive model requires no prior knowledge about the data to be compressed while the non-adaptive model must have a prior idea about the distribution of characters/phrases in the data to be compressed. Another important issue is whether to use **metadata compression** or not. Metadata, like the data, can be compressed or can be left in the uncompressed state. In this study, metadata are left uncompressed. Firstly, the size of the metadata is usually much smaller than the actual data. Secondly, metadata is referred to by all queries. So if metadata is kept compressed, it has to to decompressed very frequently leading to excessive CPU overhead. So, the database designer is faced with a much more complex problem than file compression and has to look carefully at the various issues and tradeoffs before designing a successful scheme for database compression.

For our experiments, we chose three relations. Two of these were from real-life databases: one from the NCSI database and the other from the SBI database. The first relation has character strings for all its attributes, while the second has mostly character string attributes and some numerical attributes. So, we generated a third relation, synthetically, having mostly numerical attributes. This ensured that all types of possible relations were represented by the sample relations. The different compression algorithms were applied to the three relations at file, page, record and attribute levels. The percentage compressions obtained shows that amount of compression falls as one moves from file level to attribute level primarily due to pointer overhead. However, from the query processing point of view, attribute level compression is found to be the most suitable because minimum compression/decompression is required. The study further reveals that although adaptive models may be useful for file or page level compression they are unsuitable for record level or attribute level compression. This is because, the lengths of records/attributes are, in general, not long enough for adaptive algorithms to *adapt*. So, for record or attribute level compression non-adaptive model turns out to be the only choice. Finally, after taking into account all these issues, we have suggested a column-sensitive attribute level compression scheme that performs well for all the relations we have considered. We have also studied the impact of such a scheme on all types of relational operators. Before we explore the different aspects of database compression in detail, we will highlight the merits and demerits of database compression. This will provide the motivation behind our study.

1.2 The Benefits

Database compression, when used with other conventional tools like clustering, indexing, hashing and query optimization, can enhance database performance significantly by improving **I/O performance**, **query processing speed** and **database security**. The enhancements in I/O performance and security are applicable to file compression as well. But the query processing benefits are unique to databases.

The most obvious benefit of data compression is the reduced amount of disk space required to store a given data set. This has a number of ramifications on I/O performance. Firstly, the reduced data set fits into a smaller physical disk area. Therefore, seek distances and hence seek times are reduced. Secondly, more data fits into each block,track and cylinder, allowing more intelligent clustering of related objects into physically near locations. Thirdly, compressed data can be transferred faster from the disk. In other words, data compression helps to increase disk bandwidth, by increasing the *information density* of the transferred data and not by increasing the the actual disk transfer rate. This is useful in overcoming I/O bottleneck found in many high-performance database management systems. Fourthly, the cost of network I/O is reduced. Because, in distributed databases and client-server applications, compressed data can be transferred faster across the network than uncompressed data due to lower bandwidth requirements. Finally, retaining data in the compressed form in the I/O buffer allows more records to remain in the buffer, thus increasing the buffer hit rate and reducing the number of I/Os.

The last three points are actually more general. They apply to the entire storage hierarchy of tape,disk,local and remote main memories and caches. If storage space at all levels is used more efficiently, bandwidth is saved when moving data up or down in the hierarchy, and by achieving a higher hit rate at each level. Reducing the amount of bus traffic in shared-memory systems might also allow higher degree parallelism without bus saturation.

Database compression has significant impact on query processing. This aspect is in fact, specifically related to database compression because ordinary file compression applications seldom need to process compressed data. All they do is to compress the files as a whole while storing them and decompress them as a whole when they are required later. The sole objective is to achieve a good compression ratio. But in databases an equally, if not more, important objective is to provide an efficient query processing mechanism. Compression, if used judiciously, can significantly improve query processing. On the other hand, careless use of compression can prove to be counterproductive. The potential advantages can be enumerated as follows: Firstly, the increased buffer hit rate improves the response time of transactions by reducing the number of disk accesses. Secondly, when data is stored compressed, the entire relation is stored in a much smaller number of blocks than the uncompressed relation. So when the relation is scanned during query processing, a smaller number of blocks has to be accessed. Thirdly, on compression the data and log records become shorter, so the I/O to log devices decreases. Thus, compressing database records improves I/O performance on both the primary database and the log.

Finally, database security increases to some extent as a result of database compression. This feature is actually a side effect of data compression. In order to use or interpret compressed data, one usually has to restore the information to its uncompressed form. To do this, a decoding algorithm along with the decoding key/table must be available. Even with the same coding/decoding algorithm, different files may have different decoding keys. These keys can be kept separate from the compressed files to restrict access of proprietary information to authorized users. It is however important to emphasize that data compression only enhances data security, and systems requiring stringent security/protection measures should use special techniques like cryptography to achieve that purpose.

1.3 The Drawbacks

In the last section, it has been shown that rich benefits can be expected from database compression. But, as mentioned earlier, data compression can be counter-productive if used carelessly. So the database designer has to be aware of the possible pitfalls and try to avoid them as far as possible. Firstly, the overheads involved must be considered.

They are basically of two types, namely **CPU overhead** and **storage overhead**. The CPU overhead is the overhead incurred in compressing compressing/decompressing data while query processing. In most databases updates are less frequent than searches, so the decompression process has a greater impact on the performance than the compression process. Fortunately, for most compression schemes, encoding is slower and more complicated than the decoding counterpart. So in most cases software compression of database is feasible if a little care is taken to do as little compression/decompression as possible. The storage overhead comes from the need to store the decoding keys or tables. But in general, this space is negligible compared to the space savings achieved by compression. Therefore, it is not really a serious impediment. Secondly, there is the problem of **variable record lengths**. On compression fixed sized records become variable sized. This uncertainty in the output length creates difficulties during inserts, updates and deletes. As a result the routines manipulating the records become more complex. Thirdly, most data compression schemes **disrupt data properties** that may be important for some applications. For example, by not preserving the lexical order of the compressed data, efficient sorting and searching on range predicates may become difficult. In order to support all these queries without decompressing data at all, order preserving compression schemes have to be used which may give rise to other complexities. Finally, it is obvious that reducing redundancy in data representation will in turn reduce the **ability to recover from errors**. A single bit error in the output of the Huffman's method, for example, can cause the decoder to misinterpret all subsequent bits.

1.4 Conclusion

Database compression comes with its share of advantages and disadvantages. Although at first glance the impediments may seem discouraging, some of them are no longer a serious obstacle. Now-a-days very powerful processors are available even on desktop computers. So the bottleneck due to compression/decompression and complex routines for handling variable sized records has been eliminated to a large extent. Moreover, the problem of errors and resultant catastrophic failures has been alleviated by the increasingly reliable technology in disk storage and communication media. With the proliferation of data in recent years and with the current state-of-the-art technology, data compression can become an useful and effective instrument to enhance database performance.

In the rest of this thesis, we will probe deeper into the issues related to database compression. In Chapter 2, the various compression techniques that can be used for database compression are described. In Chapter 3, the design issues involved in database compression are discussed in detail. Chapter 4 shows how data compression can be incorporated into a RDBMS and its implications. In Chapter 5, the results of the various compression algorithms for three sample relations are presented and a detailed analysis of the results is performed. Finally, in Chapter 6, the impact of database compression on query processing is explored in detail for all types of relational queries.

Chapter 2

Compression Techniques

2.1 Introduction

A lot of research has been done to develop efficient data compression techniques. In this chapter, we will explore some of these compression techniques which are suitable for compression of textual databases and are expected to yield good compression.

Data compression techniques can be divided into two major families: **lossy** and **lossless**. Lossy data compression concedes a certain loss of accuracy in exchange for greatly increased compression. Lossy compression proves effective when applied to graphic images and digitized voice. Most lossy compression techniques can be adjusted to different quality levels, gaining higher accuracy in exchange for less effective compression.

Lossless compression consists of those techniques which are guaranteed to generate the exact duplicate of the input data stream after a compression/decompression cycle. In applications like compression of text files, source codes, executables etc., the loss of even a single bit could be catastrophic. Since we are primarily interested in database compression we will concentrate only on lossless techniques. However, it should be kept in mind that with the emergence of Multimedia databases, which stores video, images, audio in addition to text, lossy compression techniques will also become a part of database compression.

2.2 Compression Models

Lossless data compression is generally implemented using one of the two different types of modeling: **statistical** and **dictionary-based**. Statistical modeling reads in and encodes a single symbol at a time using the probability of that symbol's appearance in the data. On the other hand, dictionary-based compression schemes read in input data and look for a *string* of input characters that appear in the dictionary. If a string match is found, an index into the dictionary corresponding to the matched string is output in place of the entire character string. The longer the match, the better the compression ratio. For the purpose of this study we have considered three most effective and popular schemes, namely the Huffman and Arithmetic coding which uses statistical model and LZW coding which is a dictionary-based technique. For the sake of completeness, we have also considered Run-length encoding (RLE), which does not fall into any of the above categories but simply takes advantage of successive repetitions of characters.

2.3 Runlength Coding

The Runlength coding takes advantage of consecutive repetitions of the same character. For example, consider the string "cccccc" consisting of repetitions of the same character 'c' six times. If conventional 8 bit ASCII codes are used, the string requires 6 bytes to be coded because there are six characters in the string. The RLE can encode the string in 3 bytes — "255c6". In the first byte it stores a special character, namely ASCII 255, to indicate the beginning of a run. In the next byte it stores the character which is repeated. In the third byte it stores the number of times the character is repeated. In this type of RLE compression occurs only if there are runs of length greater than three. Such a case occurs in databases when null values are stored in some fields or paddings of blanks exist at the end of character string fields.

2.4 LZW Coding

The LZW algorithm was first proposed by T. Welch [Wel84]. It is a modification of an earlier algorithm proposed by Lempel and Ziv. The LZW algorithm is organized around a translation table, referred to as a **string table**. The string table maps strings of input characters into fixed length codes. The use of 12 bit codes is common. The LZW 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 in the table. K is called the extension character on the prefix string ω. The string table is initialized to contain all single character strings in the alphabet.

The LZW string table contains strings that have been encountered previously in the data being compressed. It consists of a running sample of strings in the data, so the available strings reflect the statistics of the data.

LZW uses the "greedy" parsing algorithm, where the input string is examined character sequentially in one pass, and the longest **recognized** input string is parsed off each time. A **recognized string** is one that exists in the string table. The strings added to the string table are determined by this parsing. Each parsed input string extended by its next input character forms a new string which is added to the string table. Each such added string is assigned a unique identifier, namely its code value. An example of this procedure is shown in Figure 2.1. For simplicity a three character alphabet is used, consisting of the characters a,b and c. To begin with they are assigned the codes of 1,2 and 3 respectively as shown in Table 2.1. The principal concern in implementation is storing the string table because the strings can be of variable length. To make it tractable, each string is represented by its prefix string identifier, i.e. the code of its prefix string and the extension character, so that each table entry has a fixed length. Table 2.1 includes this alternative form of the table for the compression example of Figure 2.1.

The LZW decompressor logically uses the same string table as the compressor and constructs it similarly as data is decompressed. Each received code value is translated by the way of the string table into a prefix string and an extension character. The extension character is pulled off and the prefix string is decomposed into its prefix and extension.

Table 2.1: String Tables for LZW Coding

String Table		Alternative Table	
a	1	a	1
b	2	b	2
c	3	c	3
ab	4	1b	4
ba	5	2a	5
abc	6	4c	6
cb	7	3b	7
bab	8	5b	8
baba	9	8a	9
aa	10	1a	10
aaa	11	10a	11
aaaa	12	11a	12

Figure 2.1: LZW Encoding

INPUT SYMBOLS	a	b	ab	c	ba	bab	a	aa	aaa	a
OUTPUT	–	–	—	–	—	——	-	—	—–	–
CODES	1	2	4	3	5	8	1	10	11	12

Figure 2.2: LZW Decoding

INPUT	1	2	4	3	5	8	1	10	11
CODES	v	v	v	v	v	v	v	v	v
	a	b	1b	c	2a	5b	a	1a	10a
			v		v	v		v	v
			a		b	2a		a	1a
						v			v
						b			a
OUTPUT									
DATA	a	b	ab	c	ba	bab	a	aa	aaa

This operation continues recursively until the prefix string is a single character, which completes decompression of the code. This terminal character, called here as the final character, is the leftmost character of the string, being the first character encountered by the compressor when the string was parsed out. Figure 2.2 shows the decompression of the compressed data of Figure 2.1. An update for the string table is made for each code received (except the first one). When the code has been translated, its final character is used as the extension character, combined with the prior string, to add a new string to the string table. This new string is assigned a unique code value, which turns out to be the same code that the compressor assigned to that string. In this way, the decompressor incrementally reconstructs the same string table that the compressor used. For further details [Wel84] can be referred.

2.5 Huffman Coding

In Huffman coding, the most frequent characters are assigned shorter codes and the less frequent characters are assigned longer codes. The longer codes are constructed so that the shorter codes do not appear as prefixes. Huffman method builds a decoding tree with the minimal expected external path length. If p_i is the probability of occurrence of the i^{th} character in the alphabet, and d_i is the length of the path from the root of the tree to the leaf node representing the i^{th} character, then the Huffman tree minimizes the quantity

$$\sum_{i=1}^{N} p_i d_i$$

where N is the size of the character set. The Huffman coding comes in two different flavors, namely **non-adaptive Huffman coding** and **adaptive Huffman coding**. We consider both the variations in the following two subsections.

2.5.1 Non-adaptive Huffman Coding

In non-adaptive Huffman encoding [Huf52], the frequency distribution of each of the characters in the alphabet must be known before compression begins. This can be achieved by an initial scan of the entire data set to be compressed. The scan can be costly if the data set is large. Alternatively, the data set can be sampled, i.e. a portion of the data set can be scanned according to some scheme. In this case, we do not get an exact frequency distribution of the characters. As a result, some amount of compression has to be sacrificed. Whether this tradeoff is necessary or not, depends on the particular application. For example, while compressing huge files the sampling technique may be more cost effective. However, in databases the entire data set can be scanned while loading a relation or during insertions. So the scan comes free of any additional overhead.

The Huffman tree is constructed as follows:

1. Start by creating a leaf node for each character in the alphabet. Assign to each node a weight equal to the probability of occurrence of the corresponding character.

2. Initialize V to be the set of all nodes. Repeat step 3) until $|V| = 1$.

3. Create a new node; make it the parent of the two nodes having the smallest weights in V. Remove these two nodes from V and insert their new parent into V with a weight equal to the sum of the weights of the two replaced children.

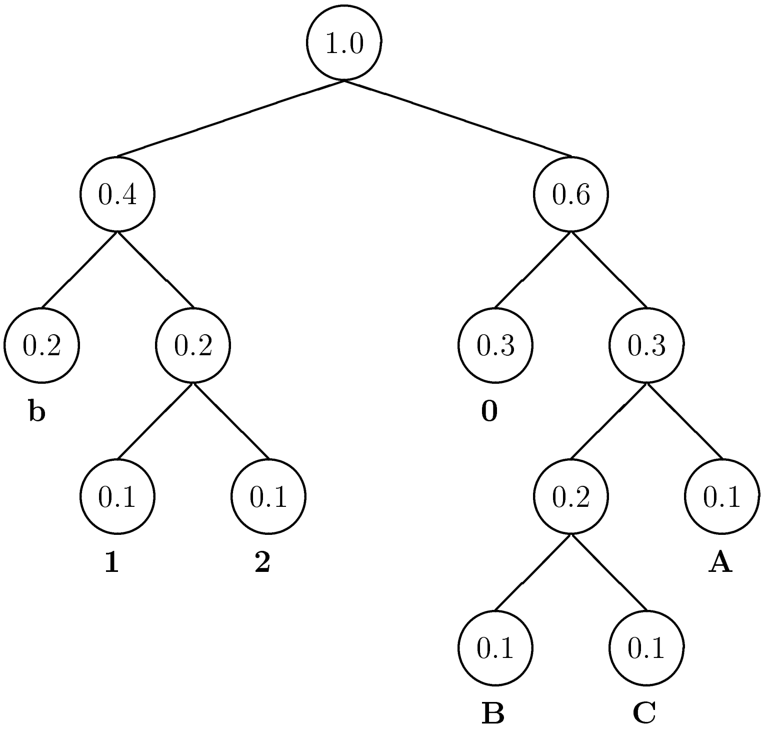
4. A Huffman tree has been generated. For each parent in this tree, label the link to one child with zero and the link to the other child as one. This labeling should be done consistently, i.e. the links to the left children should be labeled with zeros and that of the right children should be labeled with ones or vice versa. The Huffman code for a character can be obtained by concatenating the labels in the path from the root to the leaf node corresponding to that character. Note that the Huffman tree is not necessarily unique.

For a simple illustration of Huffman encoding consider the simplified character set consisting of seven characters: three letters, three digits, and the blank (denoted by b). Table 2.2 gives the arbitrarily chosen probability of occurrence of these characters and the Huffman codes resulting from the tree in Figure 2.3. The right children are labeled 0 while the left are labeled 1 in the example to generate the Huffman codes. For example, the string ABC00012bb will be coded in 27 bits using the codes given in Table 2.2 as follows

Table 2.2: A simplified character set and its Huffman code

Character	Weight	Huffman's Code
A	0.1	000
B	0.1	0011
C	0.1	0010
0	0.3	01
1	0.1	101
2	0.1	100
b(blank)	0.2	11

Figure 2.3: Huffman tree for the character set of Table 2.2



000 0011 0010 01 01 01 101 100 11 11

This is to be compared to 30 bits required if fixed length 3-bit code is used to encode the seven characters.

The decoding algorithm is simple. Given the Huffman tree, the decoder starts from the root of the tree, and goes down the tree with each received bit until it reaches a leaf node. It then outputs the character corresponding to that leaf node and starts all over again from the root of the tree. This is done until the end of the message is reached.

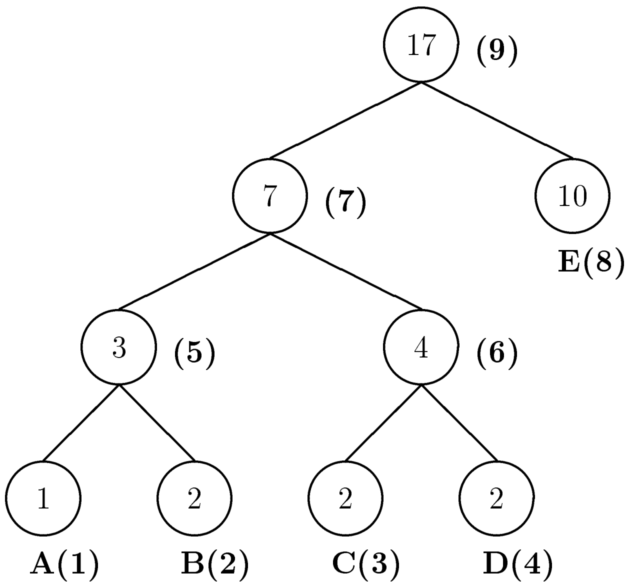
2.5.2 Adaptive Huffman Coding

The adaptive Huffman method does not require any apriori knowledge of the frequency distribution of the characters of the data set to be compressed. The scheme starts of with the assumption that all characters are equally likely to occur and adjusts the Huffman tree on the fly, based on the data previously seen. There is an easy way to take an existing Huffman tree and modify it to account for a new character. This approach introduces a concept known as the **sibling property**. A Huffman tree is simply a binary tree that has a weight assigned to every node, whether an internal node or a leaf node. Each node, except the root, has a sibling. The tree exhibits sibling property *if the nodes can be listed in order of increasing weight and if every node appears adjacent to its sibling in the list.*

A binary tree is a Huffman tree if and only if it obeys the sibling property. Figure 2.4 shows a Huffman tree that illustrates how this works. In this tree, the nodes have been assigned numbers, with the numbers assigned from left to right starting at the lowest row and working upwards. This tree has been created using the Huffman algorithm given the weights A=1, B=2, C=2, D=2, and E=10. In Figure 2.4, the A,B,C and D nodes at the bottom of the tree are numbered in increasing order starting at 1. Nodes 5 and 6 are the first two internal nodes, with weights of 3 and 4. The node numbers work their way up to node 9, the root. This arrangement shows that the tree obeys the sibling property. The nodes have been numbered in order of increasing weight, and each node is adjacent to its sibling in the list.

The adaptive algorithm can be stated as follows:

Figure 2.4: Initial Huffman tree

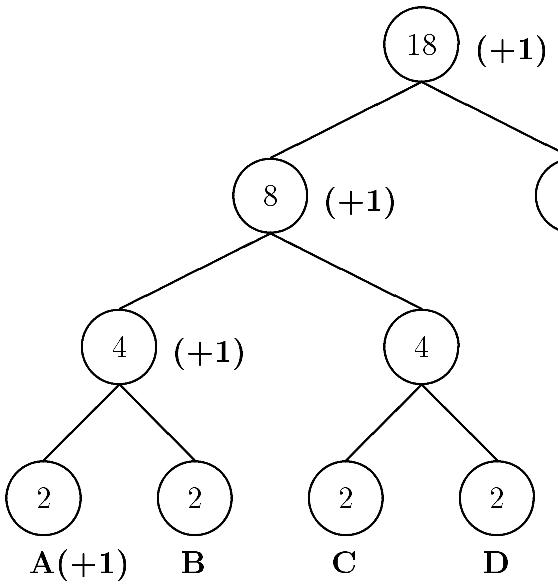


1. Build the initial Huffman tree with a predetermined frequency distribution of the character set. It is usually assumed that all characters are equally likely to occur.

2. Read an input character and increment the frequency count of that character by 1. Then move up to the parent node. Since the weight of the parent node is the sum of the weights of its children, incrementing its weight by one will adjust it to its correct value. This process continues all the way up the tree until the root is reached. It can be easily observed that the average number of increment operations required will correspond to the depth of the Huffman tree. If at any stage during the increment phase the sibling property is violated, move to step 3. Otherwise, repeat step 2.

3. This step is executed only when the sibling property is violated and occurs when the node being incremented has the same weight as the next higher node in the list. The node is detached from its present position in the tree and swapped with a node further up the list. This is done as follows. If the newly incremented node now has a weight of W+1, the next higher node will have a value of W. There may be more nodes after the next higher one that have a value of W as well. The swap phase

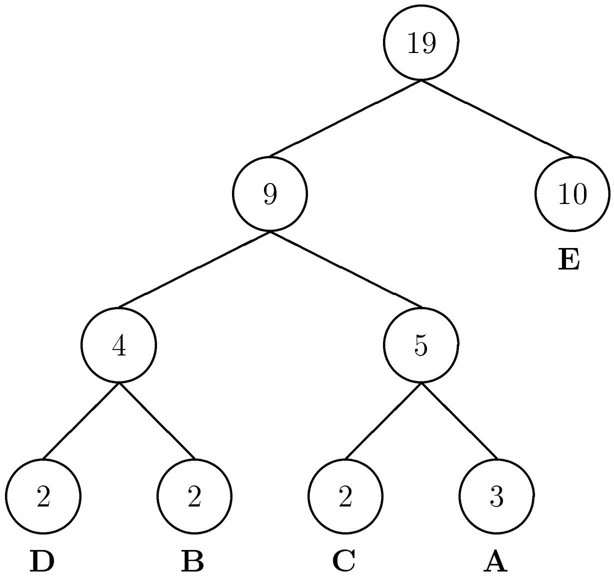
Figure 2.5: Huffman tree after node A is incremented by 1



moves up the node list until it finds the last node with weight W. That node is then swapped with the newly incremented node with weight W+1. The new node list will have a series of one or more weight W nodes, followed by the newly incremented node with weight W+1. After the swap is completed, the increment continues as in step 2. The next node to be incremented will be the new parent of the incremented node.

Figure 2.5 shows how the increment works its way up the tree when the count of node A is incremented by one. In Figure 2.6, node A was incremented from a weight of 2 to 3. Since the next node in the list, the node B, has a weight 2, the tree no longer obeyed the sibling property. This meant is was time to swap. The swap phase works its way up the list of nodes until the last node with weight 2 , node D, was found. The nodes A and D were then swapped yielding a correctly ordered tree. The swap shown in Figure 2.6 does not have a noticeable effect on the coding of the characters. The A and D nodes were swapped, but the length of their codes did not change. They were both three bits long before and after the swap. Figure 2.7 shows what happens to the tree after the A character has been incremented two more times. After the second increment, the a node

Figure 2.6: Huffman tree after node A is incremented by 2



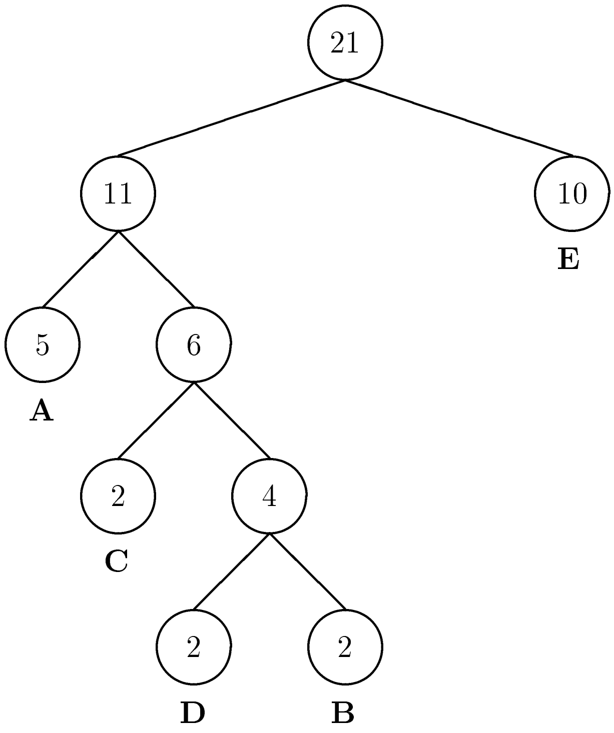
has increased enough to swap positions with an internal node at a higher level in the tree. This reshapes the tree, changing the length of the codes in the process. Now A is encoded in 2 bits only, C is still encoded in 3 bits, but B and D have slipped down to 4 bits.

The decoding process is similar to the non-adaptive one, with the difference that the Huffman tree is updated after each character is decoded. The decoder decodes correctly because at each step, it sees the same Huffman tree as that of the encoder. [Knu85] gives an efficient implementation of adaptive Huffman coding.

2.6 Arithmetic Coding

In Huffman coding, each character in the alphabet must translate into an integral number of bits after encoding. In other words, Huffman codes are integral number of bits long. This may create problems at times. For example, let the probability of occurrence of a character be 1/3. Then, from the definition of entropy, the optimal number of bits to code it is around 1.6. But Huffman coding has to assign two bits to code it, and this leads to a longer compressed data than the theoretically possible 1.6 bits. This nonoptimal

Figure 2.7: Huffman tree after node A is incremented by 3



coding becomes a noticeable problem when the probability of a character is very high. If the probability of occurrence of a character is 9/10, the optimal coding size would be 0.15 bits. The best that Huffman coding can do is to assign a 1 bit code to it, which is six times longer than necessary. This is where Arithmetic coding comes in. Arithmetic coding bypasses the idea of replacing an input symbol with a specific code. It replaces a stream of input characters with a single floating point number.

In Arithmetic coding, a message is represented by an interval of real numbers between 0 and 1. As the message becomes longer, the interval needed to represent it becomes smaller, and the number of bits needed to specify the interval grows. Successive characters in the message reduce the size of the interval in accordance with the symbol probabilities generated by the model. The more likely symbols reduce the range less than the unlikely symbols and hence add fewer bits to the compressed message.

Before anything is encoded, the range of the message is the entire interval [0,1). As each symbol is processed, the range is narrowed to that portion of it which is allocated

Table 2.3: Restricted character set and their probabilities

Symbol	Probability	Range
SPACE	1/10	[0.00,0.10)
A	1/10	[0.10,0.20)
B	1/10	[0.20,0.30)
E	1/10	[0.30,0.40)
G	1/10	[0.40,0.50)
I	1/10	[0.50,0.60)
L	2/10	[0.60,0.80)
S	1/10	[0.80,0.90)
T	1/10	[0.90,1.00)

to the symbol. To illustrate the process, let us consider the restricted alphabet and the symbol probabilities as shown in Table 2.3.

Let the message to be encoded be "BILL GATES". Each character is assigned the portion of the 0 to 1 range that corresponds to its probability of occurrence. It should be noted that the characters "owns" everything upto, but not including, the higher number. So the letter T has the range 0.90 to 0.999... The most significant portion of an arithmetic-coded message belongs to the first symbol, B, in the message "BILL GATES". To decode the first character properly, the final coded message has to be a number greater than or equal to 0.20 and less than 0.30. To encode this number, we must track the range it could fall in. After the first character is encoded, the low end for this range is 0.20 and the high end is 0.30.

During the rest of the encoding process, each new symbol further restricts the possible range of the output number. The next character to be encoded, the letter I, owns the range 0.50 to 0.60 in the new subrange of 0.2 to 0.3. So the new encoded number will fall somewhere in the 50^{th} to 60^{th} percentile of the currently established range. Applying this logic further restricts the number to 0.25 to 0.26. The algorithm that accomplish this is as follows:

Step0: Initialize: low = 0. 0, high = 1. 0;

Step1: c = next input character;

If end of input:

Table 2.4: Encoding Steps

New Character	Low Value	High Value
	0.0	1.0
B	0.2	0.3
I	0.25	0.26
L	0.256	0.258
L	0.2572	0.2576
SPACE	0.25720	0.25724
G	0.257216	0.257220
A	0.2572164	0.2572168
T	0.25721676	0.2572168
E	0.257216772	0.257216776
S	0.2572167752	0.2572167756

```
            goto Step2.
        Else
            range = high - low;
            high = low + range*high_range(c);
            low = low + range*low_range(c);
            go to Step1;
Step2:  output low;
```

Following this process for the message "BILL GATES" results in Table 2.4. So the final low value will uniquely encode the message "BILL GATES".

Given this encoding scheme, it is not very difficult to see how the decoding process works. The first symbol in the message can be found by looking at which symbol owns the space in which the encoded message falls. Since 0.2572167752 falls between 0.2 and 0.3, the first character must be a B. Then B must be removed from the encoded message to get the next character. Since the low and high ranges of B are known, their effects can be removed by reversing the process that put them in. First, the low value of B is subtracted from 0.2572167752, giving 0.0572167752. Then it is divided by the range of B, or 0.1. This gives the value of 0.572167752. This number is in the range of the next letter, which can be seen to be I in this case. The algorithm that accomplishes this is as

Table 2.5: Decoding Steps

Encoded Number	Output Symbol	Low	High	Range
0.2572167752	B	0.2	0.3	0.1
0.572167752	I	0.5	0.6	0.1
0.72167752	L	0.6	0.8	0.2
0.6083876	L	0.6	0.8	0.2
0.041938	SPACE	0.0	0.1	0.1
0.41938	G	0.4	0.5	0.1
0.1938	A	0.2	0.3	0.1
0.938	T	0.9	1.0	0.1
0.38	E	0.8	0.9	0.1
0.8	S	0.8	0.9	0.1

follows:

Step0: number = input code;

Step1: symbol = symbol whose range straddles this number;

output symbol;

range = high_range(symbol)-low_range(symbol);

number = number - low_range(symbol);

number = number / range;

Repeat Step1 until end of message is reached.

In the above algorithm, the problem of how to decide the end of message has been ignored. This can be handled either by encoding a special end-of-file symbol or by knowing the length of the encoded message. The decoding algorithm for the message "BILL GATES" will proceed as shown in Table 2.5. In summary, the encoding process is one of narrowing the range of possible numbers with each new symbol. The narrowing of the range is proportional to the probability attached to the symbol. Decoding is the inverse procedure, in which the range is expanded in proportion to the probability of each symbol that is extracted. For details of implementation please refer [Wit87, Jon88].

2.6.1 Non-adaptive Arithmetic Coding

In non-adaptive Arithmetic coding, the symbol frequencies remain fixed. An initial scan or sampling of the data generates the frequency distribution of the characters in the data. The encoder uses this distribution to compress the given data.

2.6.2 Adaptive Arithmetic Coding

In adaptive Arithmetic coding, the encoder starts off with the notion that all characters are equally likely to occur in the data to be compressed. The encoder reads in each character, outputs the code for the character, and updates the frequency distribution to account for the character just encountered. In this way the frequency distribution gets updated after each input character is read. The decoder decodes correctly because it too starts off with the assumption that all characters are equally likely to occur, and updates the distribution after each character is decoded so that it sees the same frequency distribution as the encoder at each step.

2.7 Conclusion

The basic compression algorithms have been explained in this chapter. In the next chapter, various design issues specific to database compression will be considered. After having done that, we will be ready to see how compression can be integrated with a DBMS and its implications on database performance.

Chapter 3

Design Issues

3.1 Introduction

The primary objective of ordinary file compression is to achieve a high degree of compression so that substantial savings in disk storage can be achieved. In databases too, a good compression ratio is desirable, not only because it saves disk storage, but because it improves the efficiency of query processing by reducing disk seek times, by facilitating clustering, by increasing effective disk transfer rate and by increasing the hit rate in main memory data buffers. All these issues has been discussed in Chapter 1. Another important element of query processing in databases, a feature absent in file compression, is comparison of data during query processing. This gives rise to additional design complexities which will be considered in this chapter.

3.2 Hardware Versus Software Compression

Hardware compression is much faster as compared to software compression. So the compression and decompression overheads involved in processing compressed data is much smaller. But in practice, not many hardware vendors provide compression support. Only recently, on realizing the potential of compression as an instrument in improving database performance, IBM is planning to provide compression support in their Mainframes for their DB2 Database Management System. Moreover the techniques used for hardware implementation must not be very complex. Otherwise the circuitry involved becomes too complicated. Last but not the least, the hardware implementations are vendor specific and rigid.

A more practical and cost-effective approach is to implement compression algorithms in software. This gives the flexibility to adopt techniques that are best suited for the application at hand. The only thing that must taken care of is to do as little compression and decompression as possible while processing compressed data. This ensures that the processing overhead does not become intolerable. This issue is more critical in database compression than in file compression, because of the frequent need to compress/decompress data during query processing.

3.3 Adaptive Versus Non-adaptive Model

The compression algorithms can work with two types of underlying models. One of them is the adaptive model. In this case the models have no apriori knowledge of the frequency of occurrence of the different characters in the data or the commonly occurring phrases in the data. The algorithm starts off with the assumption that all characters or all phrases are equally likely to occur in the data to be compressed. Then as the data is scanned it tries to learn about the actual distribution of characters or phrases in the data. Hence the name "adaptive". It is basically a one pass scheme in which relevant statistics is gathered and data is compressed at the same time. So adaptive schemes perform well only when the length of the data to be compressed is significant so that they can collect adequate statistics to get an idea about the actual data. But the advantage of this model is that it does not need to store any decoding keys or tables with it. On the other hand, in the non-adaptive model statistics about the frequency distribution of characters or commonly occurring phrases must be available before compression takes place. This can be achieved by simply scanning the entire data set or by sampling the data set in some fashion. Sampling gives an approximate idea about the frequency distribution or phrases in the data. Therefore, some amount of compression has to sacrificed compared to exhaustive scan of data. But sampling is much more cost-effective for some applications from the processing point of view. However, in case of database compression, the entire scan of the data comes with practically no additional cost. This is because the data can be easily scanned while loading the relation or during insertions and deletions. Non-adaptive scheme is basically a two pass procedure in which data is sampled in the first pass and the actual compression takes place in the second pass. The decoding keys or tables has to be stored with the data to enable correct decoding.

In file and page level compression in databases, as in file compression, both adaptive and non-adaptive models seem to be viable. However, in record or attribute level compression in databases, the length of the data to be compressed is too small for the adaptive models to perform well. So, one has to be more careful in the choice of the compression scheme in case of database compression than in file compression. Otherwise, even expansion of data may occur, as will be seen in the results presented in Chapter 6.

3.4 Unit of Compression

In a relational database system, a database consists of a number of relations. Each relation relation is usually stored in a separate file. Such a file consists of a number of fixed sized pages. Each page has a page header and a number of fixed sized records. The header keeps track of book-keeping informations like the number of records in a page, the vacant slots in a page etc. Each record, in its turn, consists of a number of attributes, each attribute having a fixed size. With this structure of a relation in mind, 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 (which is assumed to be stored in a file) in a database is compressed as a whole. Since a relation is generally large, adaptive models which requires considerable data length to perform well,should yield good compression ratios. But as mentioned earlier, this is not the only objective of database compression. In fact, a more serious issue in databases is efficient data processing. In case of file level compression the relations have to decompressed/compressed each time they are manipulated. This makes file level compression practically infeasible for database compression. Going down to the **page level**, relations can be compressed at the page level. Since the page size in most commercial databases vary between 1K-4K, adaptive models do have an opportunity to perform well. But again, the whole page has to be compressed/decompressed when data within it is referenced. Considering the frequency of such operations in a database, page level compression too, seems impractical from query processing viewpoint. The next lower level of compression is the **record level**. Since on an average record sizes vary between 100-300 ([BW94]) bytes adaptive models should fail to achieve significant compression. So non-adaptive models must be used. From the viewpoint of query processing this option seems much more feasible. During processing only the records that are operated on need to be decompressed. Since the length of a record is at most a few hundred bytes, the overhead involved should be tolerable for most applications. However, it should be observed that on compression, the fixed sized records become variable sized. In order to track these compressed variable sized records additional pointers must be stored resulting in some space overhead. This overhead is usually small compared to the compression achieved. Moreover, the performance improvement resulting from compression, should more than compensate for this overhead. The lowest possible level at which compression can be done is at the **attribute level**. Attributes fall in three major categories, namely, integers, floating point numbers and character strings. Generally the size of the integers vary from 1-4 bytes, that of floating point numbers between between 4-8 bytes and that of characters strings between 10-32 bytes ([BW94]). So there is little scope for adaptive models in this case. Non-adaptive models seems to be the only choice. The attribute level is best from the query processing point of view. It provides the possibility to compare and manipulate data without decompressing them at all. Were it not for one problem, attribute level compression would probably have been the most obvious choice for database compression. The problem is, on compression the attributes become variable sized. In order to do query processing without decompressing data, the starting point of each attribute in each record has to be tracked. The additional space overhead required for storing this information may become quite significant in some cases.

3.5 Query Types

In a relational database, the most important objective is to efficiently support various relational operators like **select**, **project**, **join** and **updates**. These operators generally involve comparisons of one or more attributes of one or more relations with a given value or with an attribute of another relation. Since the comparisons are done on individual attributes, decompression is necessary in case of file level, page level or record level compression. Only for attribute level compression can we hope to do comparisons with compressed data. Further, the comparisons can be of two types, **exact match** or **range**. For exact match comparisons, an attribute of one relation can be matched with the attribute of another relation (or a given value) if the attribute of the first relation and that of the second relation (or the given value) are compressed using the same statistics. However, in range comparisons more complications arise. The compression techniques we have studied do not preserve the lexical order of data on compression. So range comparison with compressed data becomes infeasible. Either the relevant records has to be an index via which the required records can be retrieved without any comparison or decompression is needed for comparing.

3.6 Conclusion

From the previous discussions, it is clear that file or page level compression is perhaps impractical for database compression. The most probable candidates are record level and attribute level compression. The only possible way to arrive at a definite conclusion is to apply the different compression algorithms discussed in Chapter 2 to relations from real databases at the different levels and analyze the results to get an insight into the various tradeoffs involved. This is done in the remaining chapters.

Chapter 4

Implementation

4.1 Introduction

We have implemented all the coding algorithms described in Chapter 2, namely **Run-length coding**, **LZW coding**, **adaptive Huffman coding**, **adaptive Arithmetic coding**, **non-adaptive Huffman coding**, and **non-adaptive Arithmetic coding**. The algorithms have been implemented in software using the **C Language** on **UNIX** platform. Each of the algorithms has been implemented at four different levels, namely file, page, record and attribute levels. To check that the encoding algorithms work correctly, the corresponding decoding algorithms have been implemented at all the four levels. To ensure that the compression ratios obtained reflect real-life situation, the following three relations have been used as inputs to the algorithms: a relation from the NCSI database, a relation from the SBI database and a synthetic relation. Each of the original relations is a file consisting of fixed sized pages, each of size 4K. Each page has a 12 byte slot map at the beginning of the page. This slot map is used to store book-keeping informations like the number of records in the page, the vacant slots in the page, if any etc. The rest of the page, i.e. 4084 bytes is used to store records of fixed size, the size depending on the particular relation.

4.2 The Relations

The details of the original relations used as input to the compression schemes are as follows:

The relation from the NCSI database, contains names of authors, titles, publishers etc. of various scientific publications. We will refer to it as **Pub** from now on . The relation consists of 80690 records, each record having 11 attributes which are all character strings. The size of each record is 1828 bytes. So the number of records in each page is 2 and the total number of pages in the relation is 40345. The unused bytes left per page is 428.

The relation from the SBI database, which we will refer to as **Bank** from now on, consists of 55862 records each of size 236 bytes. The number of attributes in the record is 24 out of which five were numerical attributes. Therefore, the number of records in a page of the original relation is 17 and the total number of pages in the relation is 3286. Unused bytes wasted per page is 72.

Since both the relations had predominantly character string attributes, a very common feature in commercial databases, a third file was constructed synthetically consisting of records having mostly numeric attributes. This relation will be referred to as **Syn** from now on. The relation has 9964 records, each of size 76 bytes and having 12 attributes. Therefore, the number of records per page is 53, unused bytes left per page is 56 bytes and total number of pages in the relation is 188.

Having looked at the relations themselves, let us now consider now how they were compressed.

4.3 Compression Scheme

We have used both adaptive and non-adaptive techniques for compression. In the non-adaptive schemes, the entire relation was scanned once to gather relevant statistics. Depending on these statistics, the relations were compressed. In the adaptive schemes, the algorithms gathered statistics and at the same time compressed the data depending on the statistics gathered upto that point.

An important thing to note at this point is the scope of the statistics used to compress data at any point of time. In non-adaptive case, the entire relation was 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. If we are considering file level compression, the scope of the statistics used refers to the entire file scanned upto the point. If we are considering page level compression, the scope of the statistics used refers to the portion of the page scanned upto that point. Similarly, the scope of the statistics gathered is limited to individual records and attributes in case of record level and attribute level compression respectively.

Compression was applied at the four levels. In **file level** compression, the entire relation is compressed as a whole. In **page level** compression, the individual pages of the relation, each of size 4K, was compressed separately. On compression each page became variable sized. So the individual pages must be tracked with the help of pointers in order to retrieve them at random. The compressed file, in **record level** compression, consists of fixed sized pages, each of size 4K. The individual records after compression, are placed in a page having sufficient space. Since the compressed records are of variable length, pointers to keep track of each record is also stored in the page. The records are stored packed (i.e. no space between successive records), beginning from one extremity of the page while the book-keeping information is stored starting from the other extremity of the page. As more and more records are placed in the page, the record boundary and the pointer boundary come closer. Finally, the distance between them becomes so close that no more records can be accommodated in the page. The book-keeping information includes the number of compressed records in the page, the number of bytes still available in the page and a series of entries containing the sizes of the successive records. The number of such entries equals the number of records stored in the page. In **attribute** level compression too, the compressed relation consists of a number of fixed sized pages, each of size 4K. The individual attributes of a record are compressed, concatenated to form a record, and stored in a page having sufficient space. After compression the fixed sized attributes become variable sized. So pointers must be stored to keep track of each attribute of each record in a page. These pointers are stored in the same page as the records themselves. The records and book-keeping information are stored as in the case of record-level compression. The only difference is that, the number of pointers stored is much more. It is in fact equal to the product of the number of compressed records in the page and the number of attributes in each record.

Now that we have a clear understanding of the implementational aspects, the next step is to look into the compression that can be achieved at the four levels with the different schemes and to analyze the results. This is the topic of the next chapter.

Chapter 5

Results and Analysis

5.1 Introduction

In this chapter we present the compression ratios obtained for the relations we studied, namely **Pub**, **Bank** and **Syn**. Then an analysis of the results is done which brings out the various aspects that have to be considered before doing database compression. At the end of the analysis we present a compression scheme which performs well for all the relations under consideration.

5.2 Results

In this section the percentage compression (PC) of each of the relations at each level and for each method is presented. It should be noted that the percentage compression, PC, is defined as

PC = (1 - size of compressed data/size of uncompressed data)*100 = CR*100

where CR is the compression ratio. In general, PC will be a number which is greater than zero and less than hundred. Closer the number is to 100, the better is the compression and vice versa. However, PC can be negative in some cases indicating an expansion in the data instead of the intended compression. This feature will be seen shortly in the results.

The percentage compression achieved for the relations **Pub**, **Bank** and **Syn** are shown in Tables 5.1, 5.2, 5.3 respectively. In the tables, for record and attribute level compression, we show the **average**, **maximum** and **minimum sizes** of the compressed records, as well as the **variance** in the size of the compressed records. The **average overhead** refers to the average number of bytes wasted per page to store pointers for tracking variable sized records/attributes. **Average fragmentation** refers to the amount of unused space left per page on an average, because it is to small for any record to fit in. All these parameters have no meaning in case of file or attribute level compression and therefore, only the percentage compression is shown for file or page level. The following acronyms have been used: **RLE** for Run-length 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.

5.3 Analysis

Having presented the percentage compression for the three relations, the inferences that can be drawn from these numbers have to be considered. We will look at the results from two dimensions – from the viewpoint of the compression method and from the viewpoint of the unit/level of compression.

First of all let us consider the overheads that creep in as one goes down from the file level to the attribute level. In file level compression, the file is compressed as a whole and no overhead is involved. In page level compression, each page becomes variable sized on compression. So, as has already been mentioned, pointers must be stored for random page access. The number of such pointers is equal to the number of pages in the relation. The pointers are stored separately and the space overhead involved is insignificant. In record level compression too, pointers must be stored to track the variable sized compressed records. These pointers are however, stored along with the records in the same page. The number of such pointers in each page equals the number of compressed records stored in the page. So the percentage overhead will be small if the record size is large and increases as the record size decreases. However, this should still be tolerable for most relations because the compression achieved will easily offset this overhead. Coming down to the

Table 5.1: Percentage Compression For Pub

Method	Level of Compression			
	File	**Page**	**Record**	**Attribute**
RLE	80.35	80.00	79.1 Avg. rec. sz.= 403 Min. rec. sz.= 235 Max. rec. sz.=1066 Var. rec. sz.=3649 Avg. overhead= 19 Avg. fragment= 208	77.50 Avg. rec. sz.= 413 Min. rec. sz.= 235 Max. rec. sz.=1066 Var. rec. sz.=3649 Avg. overhead= 195 Avg. fragment= 220
LZW	91.57	82.90	79.18 Avg. rec. sz.= 402 Min. rec. sz.= 259 Max. rec. sz.= 844 Var. rec. sz.= 2450 Avg. overhead= 19 Avg. fragment= 204	70.23 Avg. rec. sz.= 544 Min. rec. sz.= 39 Max. rec. sz.= 986 Var. rec. sz.= 3243 Avg. overhead= 147 Avg. fragment= 283
AA	79.45	76.10	72.40 Avg. rec. sz.= 525 Min. rec. sz.= 373 Max. rec. sz.= 980 Var. rec. sz.= 3057 Avg. overhead= 14 Avg. fragment= 264	60.00 Avg. rec. sz.= 725 Min. rec. sz.= 593 Max. rec. sz.= 1121 Var. rec. sz.= 2777 Avg. overhead= 110 Avg. fragment= 355
AH	75.58	73.95	71.80 Avg. rec. sz.= 535 Min. rec. sz.= 410 Max. rec. sz.= 950 Var. rec. sz.= 1841 Avg. overhead= 14 Avg. fragment= 272	66.50 Avg. rec. sz.= 622 Min. rec. sz.= 472 Max. rec. sz.= 983 Var. rec. sz.= 2237 Avg. overhead= 65 Avg. fragment= 314
NAA	78.82	78.70	78.01 Avg. rec. sz.= 424 Min. rec. sz.= 278 Max. rec. sz.= 995 Var. rec. sz.= 2641 Avg. overhead= 18 Avg. fragment= 216	76.57 Avg. rec. sz.= 430 Min. rec. sz.= 283 Max. rec. sz.= 1002 Var. rec. sz.= 3298 Avg. overhead= 187 Avg. fragment= 227
NAH	74.91	74.86	74.40 Avg. rec. sz.= 488 Min. rec. sz.= 375 Max. rec. sz.= 924 Var. rec. sz.= 1885 Avg. overhead= 15 Avg. fragment= 248	72.20 Avg. rec. sz.= 510 Min. rec. sz.= 396 Max. rec. sz.= 946 Var. rec. sz.= 1844 Avg. overhead= 157 Avg. fragment= 274

Table 5.2: Percentage Compression For Bank

Method	Level of Compression			
	File	**Page**	**Record**	**Attribute**
RLE	24.40	24.20	22.00 Avg. rec. sz.=182 Min. rec. sz.=153 Max. rec. sz.=218 Var. rec. sz.=322 Avg. overhead= 21 Avg. fragment= 58	11.00 Avg. rec. sz.=182 Min. rec. sz.=153 Max. rec. sz.=218 Var. rec. sz.=322 Avg. overhead=453 Avg. fragment=174
LZW	81.00	65.00	36.78 Avg. rec. sz.=148 Min. rec. sz.=126 Max. rec. sz.=179 Var. rec. sz.=270 Avg. overhead= 26 Avg. fragment= 64	-19.56 Avg. rec. sz.=249 Min. rec. sz.=148 Max. rec. sz.=281 Var. rec. sz.=451 Avg. overhead=341 Avg. fragment=199
AA	51.36	48.43	29.45 Avg. rec. sz.=163 Min. rec. sz.=147 Max. rec. sz.=184 Var. rec. sz.=333 Avg. overhead= 24 Avg. fragment=119	-6.27 Avg. rec. sz.=225 Min. rec. sz.=212 Max. rec. sz.=241 Var. rec. sz.=343 Avg. overhead=383 Avg. fragment= 98
AH	49.79	47.50	35.44 Avg. rec. sz.=151 Min. rec. sz.=132 Max. rec. sz.=188 Var. rec. sz.=187 Avg. overhead= 26 Avg. fragment= 77	-3.13 Avg. rec. sz.=216 Min. rec. sz.=181 Max. rec. sz.=253 Var. rec. sz.=325 Avg. overhead=385 Avg. fragment=126
NAA	46.41	46.35	44.87 Avg. rec. sz.=129 Min. rec. sz.=111 Max. rec. sz.=170 Var. rec. sz.=225 Avg. overhead= 30 Avg. fragment= 62	26.50 Avg. rec. sz.=149 Min. rec. sz.=126 Max. rec. sz.=191 Var. rec. sz.=230 Avg. overhead=554 Avg. fragment= 81
NAH	43.45	43.40	41.20 Avg. rec. sz.=138 Min. rec. sz.=117 Max. rec. sz.=180 Var. rec. sz.=152 Avg. overhead= 28 Avg. fragment= 68	28.47 Avg. rec. sz.=144 Min. rec. sz.=126 Max. rec. sz.=186 Var. rec. sz.=282 Avg. overhead=570 Avg. fragment= 81

Table 5.3: Percentage Compression For Syn

Method	Level of Compression			
	File	**Page**	**Record**	**Attribute**
RLE	21.54	21.0	18.60 Avg. rec. sz.= 61 Min. rec. sz.= 52 Max. rec. sz.= 76 Var. rec. sz.= 52 Avg. overhead= 65 Avg. fragment= 39	4.25 Avg. rec. sz.= 61 Min. rec. sz.= 52 Max. rec. sz.= 76 Var. rec. sz.= 52 Avg. overhead=664 Avg. fragment= 38
LZW	46.20	40.59	20.63 Avg. rec. sz.= 59 Min. rec. sz.= 49 Max. rec. sz.= 75 Var. rec. sz.=100 Avg. overhead= 66 Avg. fragment= 39	-33.33 Avg. rec. sz.= 89 Min. rec. sz.= 80 Max. rec. sz.=104 Var. rec. sz.=115 Avg. overhead=477 Avg. fragment= 56
AA	44.19	42.48	20.74 Avg. rec. sz.= 59 Min. rec. sz.= 53 Max. rec. sz.= 68 Var. rec. sz.= 60 Avg. overhead= 66 Avg. fragment= 51	**-21.20** Avg. rec. sz.= 80 Min. rec. sz.= 75 Max. rec. sz.= 86 Var. rec. sz.=130 Avg. overhead=524 Avg. fragment= 38
AH	44.02	38.67	13.83 Avg. rec. sz.= 64 Min. rec. sz.= 50 Max. rec. sz.= 87 Var. rec. sz.= 61 Avg. overhead= 39 Avg. fragment= 148	-2.66 Avg. rec. sz.= 66 Min. rec. sz.= 55 Max. rec. sz.= 88 Var. rec. sz.= 87 Avg. overhead=622 Avg. fragment= 40
NAA	44.38	44.33	40.21 Avg. rec. sz.= 44 Min. rec. sz.= 37 Max. rec. sz.= 59 Var. rec. sz.= 92 Avg. overhead= 88 Avg. fragment= 46	19.57 Avg. rec. sz.= 49 Min. rec. sz.= 42 Max. rec. sz.= 65 Var. rec. sz.= 85 Avg. overhead=786 Avg. fragment= 46
NAH	43.85	43.80	39.89 Avg. rec. sz.= 45 Min. rec. sz.= 37 Max. rec. sz.= 59 Var. rec. sz.= 28 Avg. overhead= 88 Avg. fragment= 20	20.74 Avg. rec. sz.= 48 Min. rec. sz.= 40 Max. rec. sz.= 62 Var. rec. sz.= 69 Avg. overhead=802 Avg. fragment= 41

attribute level, a pointer has to stored for each attribute of each record. In other words, the number of pointers stored in each page is equal to the product of the number of compressed records stored in the page and the number of attributes in each record. This overhead can, therefore, be significant. Even then it should be tolerable when the record size is large and the number of attributes is small. But as the record size decreases and the number of attributes in a record increases, this overhead can significantly reduce the effective compression achieved. As mentioned in Chapter 1, we will refer to this type of overhead as **pointer overhead**. Apart from this there is another type of overhead which can 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. We can say that on an average 1/2 a byte is wasted for each attribute. This overhead becomes noticeable as the record size decreases and the number of attributes increases. This overhead is also present at the record level, but it is insignificant. In the page level, it is practically nil. As mentioned earlier, this overhead will be referred to as **padding overhead**.

Keeping in mind the overheads just mentioned, let us look more closely at the performance of the different schemes.

*** RLE:** The percentage compression for RLE is unexpectedly good for **Pub**, whereas for **Bank** and **Syn** they are not so compared to the more sophisticated techniques. As mentioned earlier, most of the compression for RLE in a textual database is expected to come from runs of blanks/nulls at the end of string fields. The compression ratios for **Pub** can be explained easily by considering the statistics of data in the relation **Pub** shown in Table 5.4.

Table 5.4				
Attribute no.	**Avg.Sz.**	**Max.Sz.**	**Min.Sz.**	**Variance**
1	28	562	5	745
2	76	544	10	1306
3	52	146	9	508
4	6	10	3	11
5	5	5	5	0
6	22	44	7	165
7	7	126	4	12
8	7	13	5	12
9	9	9	9	0
10	77	190	35	362
11	76	147	9	445

It is seen that the average sizes of most of the attributes is much smaller than their maximum allocated size. 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. So, what is needed is a DBMS that support variable sized records. At the beginning, we have pointed out that handling of variable sized records may be considered as one of the main hindrances for database compression. But if variable record size is already supported in a DBMS for dealing with skewed data, then compression can be incorporated without any significant change in the record handling routines. The deficiencies of the RLE becomes apparent when the percentage compression of the other two relations is studied. It is also observed that the compression ratio declines as one goes from file level to attribute level. This is because of the pointer overhead. It should be noticed that in RLE there is no padding overhead because records/attributes are always encoded in an integral number of bytes. From this discussion, we can conclude that although RLE is simple to implement and fast, its effectiveness and use is limited in case of databases that contain textual data.

* **Adaptive Schemes:** Amongst the adaptive schemes, LZW performs better as compared to Arithmetic or Huffman coding at the file and page level. This is because that LZW can replace long strings appearing in the string table with one single code. For example, consider **Pub**. It has records of size 1828 bytes and one of its attribute is "Subject". A typical entry in this field can be "Computer Science". Once this string, "Computer Science", is included in the string table, any future occurrence of this string within the scope (the file/page in this case) will be replaced by a single code. On the other hand, the other adaptive methods scan one character after another within the file/page, as the case may be and updates the statistics gathered to adapt to the local data. Now, the individual attributes are separated by a record length. If the record length is long and data along the record is heterogeneous, these schemes may lose the picture of the data of that particular attribute and may have to re-adapt to the local characteristics. This effect is more noticeable in relations with long records like **Pub** and decreases as the record size decreases. However, LZW's performance is comparable to and worse than the other adaptive methods at the record and attribute level respectively, except for **Pub**. This is because the record/attributes sizes are in general small and so LZW does not have the opportunity to gather enough strings for its string table within the scope of a record or an attribute. The exception in **Pub** occurs because its record size or attribute sizes are unusually large. As the record size and the attribute sizes decreases in **Bank** and **Syn**, LZW's performance degrades sharply. The performance of adaptive Huffman and adaptive Arithmetic coding are more or less comparable in all the cases at all levels as can be seen from the tables.

It is seen that for adaptive schemes the compression ratio falls significantly as the unit of compression changes from file to attribute. The decline from file to page is because of the reduced scope to *adapt*, which is now confined to a single page. The significant 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 we have ended up by expanding the relations. The scope to adapt is now confined to the length of a single attribute, the pointer overhead increases, and some amount of padding overhead, although negligible compared to the other effects, creeps in. The net result is that the compression achieved is offset and the relations expand. It is observed that LZW is the worst affected, followed by adaptive Arithmetic and then adaptive Huffman. So, we can conclude 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. In databases, a major issue is efficient query processing, which restricts our choice to record/attribute level compression. So, we can conclude that adaptive schemes are not suitable for compression of databases storing textual data.

*** Non-adaptive Schemes:** The non-adaptive Arithmetic and Huffman schemes produce comparable results in all cases at all levels. It is found that at file level the adaptive policies are sometimes better than the corresponding non-adaptive policies. This is because the adaptive policies can adapt to local characteristics of the data when data pattern varies from one portion of the relation to another. On the other hand, the non-adaptive policies have a static smoothed-out global view of the data. However, as we move down to the record level and then to the attribute level, the non-adaptive schemes outperform the adaptive schemes convincingly. It is observed that the compression ratios achieved at the attribute level is significantly less than that at the record level. This is mainly because of the pointer overhead and to some extent due to the padding overhead.

From the previous discussions, it has become clear that for efficient data processing, compression has to done at the record or attribute level. This in turn requires a non-adaptive scheme. From the query processing viewpoint attribute is the ideal unit of compression. But from the results we have, it seems that the pointer overhead will tilt the balance in favor of record level compression. So we have come up with an alternative scheme. We have used the non-adaptive Arithmetic coding and have taken into account the peculiarity of data organization in a database. In a relation, data belonging to the same attribute (column) is semantically related and is expected to have similar distribution. So instead of having a single frequency distribution for the entire relation, we suggest that there should be a frequency distribution table for each attribute in a relation.

Table 5.5: Percentage Compression	
Relation	**Col. Sensitive Attr. Level NAA Coding**
Pub	79.54 Avg. rec. sz. = 375 Min. rec. sz. = 239 Max. rec. sz. =1088 Var. rec. sz. =3433 Avg. overhead = 215 Avg. fragment = 203
Bank	44.21 Avg. rec. sz. =108 Min. rec. sz. = 95 Max. rec. sz. =151 Var. rec. sz. = 69 Avg. overhead =731 Avg. fragment = 66
Syn	21.27 Avg. rec. sz. = 48 Min. rec. sz. = 39 Max. rec. sz. = 70 Var. rec. sz. = 75 Avg. overhead =807 Avg. fragment = 37

This is expected to reflect the characteristics of each attribute more correctly and prevent any smoothing out of the peculiarities of a particular attribute as may happen in the case of a single table. This scheme was tested for each of the relation. The results obtained are in table 5.5.

The compression ratios show a significant improvement over the attribute level compression with a single table, except in case of **Syn** where the improvement is only marginal. For **Pub** it is better than even the file level compression achieved with a single table. For **Bank** it is equal to that achieved at the record level. In **Syn** however, this scheme fails to match the record level compression. This is because that almost all the attributes are numbers which are similarly distributed and no additional semantics is captured by maintaining tables for each column. However, in general such occurrences are rare and the individual columns in a relation usually have characteristics of its own which is different from that of the other columns in the relation. In such a case maintaining multiple tables will definitely be of significant benefit as observed in cases of **Pub** and **Bank**. In these cases, the raw compression, achieved by maintaining a table for each column, is much more; so much so that the pointer overhead and padding overheads are offset.

Since attribute level encoding is best suited for query processing and since the scheme suggested above is expected to give results that are, on an average, comparable with record level compression, we can conclude that column-sensitive attribute level compression is the most viable approach to database compression. In the next chapter, we will study the implications of attribute level compression on query processing in detail.

Chapter 6

Query Processing

6.1 Introduction

One of the principal goals of a DBMS is to do efficient query processing. As pointed out in Chapter 1, database compression is not merely a means of saving storage, it has significant impact on query processing as well. Conventional performance enhancement techniques like indexing, hashing, query optimization etc. produce significant performance improvement, mainly by reducing the number of disk accesses during query processing which is finally reflected in terms of reduced response time of transactions. If in addition to these techniques, data compression can be used effectively, the number of disk accesses can be farther reduced and farther improvement in response time can be achieved. Better the compression ratio, greater is the reduction in disk accesses and more pronounced is the decrease in response time of transactions. In the previous chapter, we have described a column-sensitive attribute level coding scheme which yields acceptable compression ratios for all the relations studied. In this chapter, the tradeoffs involved in query processing on a database compressed at the attribute level is discussed.

Before considering the various relational operators like **select**, **project**, **join**, **insert**, **delete**, etc., we will look at some of the notations used in the analysis. For an uncompressed relation R:

- the number of tuples in R is T_R.

- the number of blocks in which these tuples fit when stored packed is B_R. In general, B_R is much smaller than T_R. In fact, n = T_R/B_R, where n is the number of tuples that can fit in a block.

- $I_{R.A}$, or simply I_A when R is understood, is the expected number of different values of attribute A that is found in R.

Farther, for the compressed relation R:

- the number of blocks in the compressed relation is B_{R_c}.

- the average number of tuples in a block of the compressed relation is n_c. The term "average" is used because the size of a compressed tuple is not fixed and so the number of tuples that can be fitted into a block varies.

The following assumptions are also made to facilitate our analysis:

- It is assumed that all the A-values are equally likely to occur in a relation. Therefore the number of tuples having a particular value of attribute A in relation R is $T_R/I_{R.A}$.

- the indices are not compressed. This ensures faster index lookup in case of range queries. This is because the compression techniques that have been used do not preserve lexical ordering. So, decompression must be done for comparison during index search if the indices are kept compressed.

- the cost of accessing index blocks is ignored. This is because index is kept uncompressed. Hence, for a particular query, the number of index block accesses will be the same irrespective of whether the database is compressed or not. Moreover, the cost of accessing index blocks, for most queries, is very small compared to the cost of accessing the data blocks.

- the decompression during query processing is **complete**, i.e., all the tuples in the final result is decompressed. However, we would also like the decompression to be **precise**. In other words, ideally, we would like to decompress only those tuples that appear in the final result. This ensures minimum decompression during query processing.

- for the purpose of illustration, a percentage compression of 50% is assumed. Results obtained in the previous chapter show that for the attribute level scheme suggested, the compression percentages vary from 21-78 percent for the three relations studied. So, percentage compression of 50% does not seem to be unrealistic [BW94, Grf93]. A compression percentage of 50 percent is assumed which implies $B_{R_c} = B_R/2$.

- the page size is equal to the size of a disk block. This implies that fetching a page from disk involves one disk access.

- the **total cost** of a relational operation will be expressed in terms of the number of disk block accesses required for that operation, because it is, by far, the costliest operation during query processing. So, compression/decompression overhead, if any, will be normalized to an equivalent number of disk accesses, assuming that each disk access takes around 10ms [CL88].

- the average size of compressed floating point or integer attributes is about 2 bytes and that of a typical character string attribute is about 10 bytes, assuming around 50% compression. Now, to decompress a byte about 15 machine instructions are required [BW94]. Even for PCs, one instruction cycle takes much less than $1\mu sec$. So, we can safely assume that, in the worst case, the time required to decompress a byte, t_{db}, is $1\mu sec$. So, the overhead involved in decompressing an attribute of a compressed relation R is

  $$T_R l_a t_{db} \tag{6.1}$$

  where l_a is the average size of the compressed attribute. This cost, expressed in terms of number of disk accesses, becomes

  $$T_R l_a t_{db}/0.01 \tag{6.2}$$

  where t_{db} is in seconds. Therefore, for the sample relation R having 100000 tuples, the cost is 20 disk accesses for decompression of a numerical attribute while it is around 100 disk accesses for a character string attribute. This result will be used later in the chapter.

- we introduce a new dependence, called the **compression dependence** of the attributes. Two attributes, of the same relation or of two different relations, are said to be compression dependent, if they are compressed using the same statistics. Attributes which are compression dependent can be matched without any decompression and is thus very useful for processing equijoins. It may apparently seem that compressing two attributes using the same statistics, may lead to loss in compression. But it is usually not the case. Firstly, a join is performed between two attributes that have the same data type. Joining a character string attribute of one relation with an integer attribute of another relation, does not make sense in general. Secondly, the two attributes that are joined usually have some strong semantic correlation. For example, the join attributes can be either both names of persons, or names of companies, or names of publishers. But it is unlikely that we would join the names of persons with the names of companies. So in practice, it is enough to have compression constraint for only those attributes that have the same data type and which are semantically related for the join to make sense. These attributes, therefore, will have similar distribution of characters. So, using the same statistics does not result in compression loss. It should be noted that it is up to the database designer to specify the compression constraints, wherever required. If two attributes which are not compression dependent are joined, decompression of the join attributes is necessary for matching them. In the following discussion, we assume that the join attributes are compression dependent.

For the purpose of illustrating the improvement in performance due to compression, two hypothetical relations R and S will be used throughout this chapter. R is used for single relation operations like select, project etc. while both R and S are used for joins. The relations are assumed to be as follows:

- For relation R, $T_R = 100000$,$B_R = 10000$, $B_{R_c} = 5000$.

- For relation S, $T_R = 100000$,$B_S = 5000$, $B_{R_c} = 2500$.

Having described the notations and assumptions that will be used for the analysis, we will proceed to discuss the effect of database compression on each of the relational operators.

6.2 Selection

Selection on a single attribute is considered in this section. Extension to selection on multiple attributes is not difficult. Selection done on a single attribute can be of two types:

- **exact match selection of the form $\sigma_{A=c}(R)$:** those records in relation R are selected for which attribute A has the value c.
- **range selection of the form $\sigma_{r_1 \le A \le r_2}(R)$:** those records in relation R are selected for which the value of attribute A is greater than equal to r_1 and less than equal to r_2.

In the following two subsections, these two types of selection queries are studied.

6.2.1 Exact Match Selection

In exact match selection, the cost of retrieving the desired tuples depend on whether an index is available on the selection attribute A or not, and if so on the type of the index. This gives rise to the following three scenarios:

1. **A clustering index on A exists:**
   Since, it is assumed that all A values are equally likely to occur, the number of tuples selected is $1/I_A$ of the total number of tuples in R. Since the relation is ordered on attribute A, these T_R/I_A tuples must be on stored consecutive blocks. The number of blocks required is, therefore, approximately B_{R_c}/I_A. The first of these blocks can be accessed through the index directly and the rest are accessed sequentially thereafter. Since the blocks are accessed via the index, only the tuples having the desired value of A are retrieved. As a result, no comparison is required and no question of decompression arises. It should be noted that, if $B_{R_c} < I_A$, then only

one disk access is required to retrieve the desired tuples. So, in this case, number of disk block accesses is

$$max(1, B_{R_c}/I_A) \tag{6.3}$$

and no decompression is necessary.

Let us assume two possible values of I_A, namely $I_A = 100$ and $I_A = 10000$. When $I_A = 10000$,

total cost for the uncompressed relation = 1 disk access.

total cost for the compressed relation = 1 disk access.

i.e., the costs are the same. However, when $I_A = 100$,

total cost for the uncompressed relation = 100 disk accesses.

total cost for the compressed relation = 50 disk accesses.

a savings of 50%. So, upto 50% savings can be expected in this type of queries, if a compression of 50% is achieved.

2. **A non_clustering index on A exists:**

   The number of tuples selected is, as in the previous case, T_R/I_A. However, now the relation is not ordered on attribute A. So the required tuples can be located on arbitrary pages of the compressed relation. In the worst possible scenario, all the tuples can be located on separate pages, resulting in T_R/I_A block accesses. However, if $T_R/I_A \geq B_{R_c}$ – the number of pages in the compressed relation, then only B_{R_c} blocks need to be accessed. The blocks are accessed directly through the index. So, comparison is unnecessary and the question of decompression does not arise, as in the case of clustered index. Here, number of disk block accesses is

$$min(B_{R_c}, T_R/I_A) \tag{6.4}$$

   and no decompression is necessary.

   Since the quantity T_R/I_A does not change due to compression, the number of block accesses in this case, is usually not effected by compression. However, there are benefits when $T_R/I_A \geq B_{R_c}$. To illustrate the idea, two possible values of I_A are chosen, $I_A = 10000$ and $I_A = 10$. When $I_A = 10000$, $T_R/I_A = 10$. So, for both the uncompressed and compressed relation,

total cost = 10 disk accesses.

Now for, $I_A = 10$, $T_R/I_A = 10000$. So,

total cost for the uncompressed relation is min(B_R,T_R/I_A) = 10000 disk accesses.

total cost for the compressed relation is min(B_{R_c},T_R/I_A) = 5000 disk accesses.

To sum up, savings is achieved only when $T_R/I_A \geq B_{R_c}$, i.e., when $n_c \geq I_A$ – when the average number of tuples in a block is greater than equal to the number of distinct values of attribute A. This does not seem to occur very frequently in databases, except for attributes having a limited domain like departments in a company, states of a country etc. So, the benefit of compression is, to some extent, limited in this type of query. Nevertheless, there is no additional overhead if data is kept compressed.

3. **No index on attribute A exists:**

   In this case each block of the compressed relation is accessed sequentially. For each record in a block, the value of the relevant attribute is compared with the value of the compressed search key. It should be noted that the search key must be compressed using the same statistics as that used for attribute A. The starting position and the length of the attribute is required for comparison and these can be obtained from the information stored in the page of the compressed relation. In this case, number of disk block accesses is

$$B_{R_c} \tag{6.5}$$

   and no decompression is necessary.

   Therefore a 50 percent savings in disk accesses can be achieved for a percentage compression of 50%. For the example relation R,

   total cost for the uncompressed relation = 10000 disk accesses.

   total cost for the compressed relation = 5000 disk accesses.

6.2.2 Range Selection

In range selection, the tuples to be selected are those for which the value of attribute A lies in the range specified in the query. We define the fraction of the tuples of the relation that lie in the desired range as the **selectivity, s**, of the given query. The value of s depends on the range specified and the distribution of attribute A in its domain. The cost of processing the query, as in the exact match case, depends on the presence or absence of index on A and the type of the index, giving rise to the following three cases:

1. **A clustering index on A exists:**

   The number of tuples selected is $T_R s$, where s is the selectivity. Since the relation is ordered on A, all these tuples must be located on consecutive pages, the number of such pages being $B_{R_c} s$. The tuples are accessed via the index, hence no comparison and decompression is required as pointed out in the exact match case. If, however, $B_{R_c} s < 1$, the only one vlock is required. Therefore, number of disk block accesses is

   $$max(1, B_{R_c} s) \tag{6.6}$$

   and no decompression is necessary. For illustration we take an unusual extreme value of s, namely $s = 1/10000$ and a typical value, $s = 1/2$. When $s = 1/10000$,
   total cost for the uncompressed relation = 1 disk access.
   total cost for the compressed relation = 1 disk access.
   When $s = 1/2$,
   total cost for the uncompressed relation = 5000 disk accesses.
   total cost for the compressed relation = 2500 disk accesses.
   a savings of 50%. Very low selectivities like $s = 10000$ are unusual for range selections; so, for this type of query a savings of 50% can be expected for a compression of 50% in almost all cases.

2. **A non_clustering index on A exists:**

   The number of tuples selected is, as in the previous case, $T_R s$. But now they can be located in any page of the relation. So a maximum of $T_R s$ disk accesses may be required. But if $B_{R_c} \leq T_R s$, then B_{R_c} block accesses should suffice. Since the tuples are accessed via the index, no comparison and decompression is required. So, number of disk block accesses is

$$min(B_{R_c}, T_R s) \tag{6.7}$$

and no decompression is necessary.

In this case the benefits of compression manifests itself when $BR_c \leq T_R s$, i.e., when $n_c \geq 1/s$. In range selection, s typically varies from 1/2 to 1/20. This means if $n_c \geq 20$, then most range selection queries have a very good chance to of benefiting. It should be noted here, that if the page size is sufficient and compression is used, $n_c \geq 20$ is not impractical for most relations. So the possibility of performance enhancement due to data compression here, is much more than in the corresponding exact match case. For a query, $s = 1/4$ say. Then, $T_R s = 25000$. So,
total cost for the uncompressed relation = 10000 disk accesses.
total cost for the compressed relation = 5000 disk accesses.
a significant savings of 50%.

3. **No index on attribute A exists:**
As in the case of exact match selection, all the blocks of the relation are accessed sequentially. However, in all the coding techniques we have used, the lexical order changes as a result of compression. So it is not possible to compare without decompressing the attributes. Only the particular attribute concerned has to be decompressed. This is possible because the position and the length of the attribute can be found in the page itself. So, number of block accesses is

$$B_{R_c} \tag{6.8}$$

and decompression of attribute A for all tuples in R is necessary. As has already been explained in section 6.1, the cost of such a decompression is equivalent to 10 or 50 block accesses for relation R depending on whether the attribute A is integer/float or character string respectively. This additional overhead is almost always offset by the reduction in disk accesses due to compression, in even the PCs. For the relation R,

total cost for the uncompressed relation = 10000 disk accesses.

total cost for the compressed relation = 5000 + (d_cost) disk accesses.

where decompression cost *d_cost*, as we have mentioned earlier, is 20 disk accesses if A is integer/float and 100 disk accesses if A is a character string. This overhead is less than 2% of the disk accesses needed to access data and hence negligible for all practical purpose. Here too, a 50% compression ensures about 50% savings in disk accesses.

We have, therefore, shown that all sorts of selection queries benefit from database compression. The gain, in almost all cases, is directly proportional to the percentage compression achieved.

6.3 Project

In order to project certain attributes of a relation, all the blocks have to be accessed sequentially and the desired attributes have to be filtered out. The locational information of the attributes is required for processing and can be obtained from the book-keeping information in the pages themselves. So, number of disk block accesses is

$$B_{R_c} \tag{6.9}$$

and no decompression is necessary.

Therefore in projection, a 50% I/O savings takes place if a 50% compression is achieved. For the relation R,

total cost for the uncompressed relation = 10000 disk accesses.

total cost for the compressed relation = 5000 disk accesses.

6.4 Join

Join operations are much more expensive than operations on single relations like selection, projection etc. So the potential benefit from database compression can be much more significant in this case. We will introduce two more parameters at this stage which will be useful for the calculation of costs of join operations.

- **The output cost:**

  Let U stand for the number of blocks needed to store the result of the join. As the result must be written to the secondary storage, U disk block accesses are required to store the output. For joins, U often dominates the total cost. The output cost can be computed as follows [Ulm89]:

  Let $R(A,B) \bowtie S(B,C)$ represent the join of two relations R and S, and suppose that for the join attribute B, either the domain of B in R is a subset of the domain of B in S, or vice versa. Assume that each tuple in the product of the domains of all the attributes of R is equally likely to be in R, and similarly in S. Then the expected size of $R(A,B) \bowtie S(B,C)$ is

  $$T_R T_S / I_B \tag{6.10}$$

  where I_B is the size of the domain of B, in R os S, whichever is the superset of the other.

  The size of each output tuple is approximately $(l_R + l_S)$, where l_R and l_S are the average sizes of the compressed tuples in R and S respectively. So ,

  $$U = T_R T_S (l_R + l_S) / I_B b \tag{6.11}$$

  where b is the block size in bytes. Now, B_{R_c} is approximately $T_R l_R / b$, and B_{S_c} is approximately $T_S l_S / b$. Therefore,

  $$U = (B_{R_c} T_S + T_R B_{S_c}) / I_B \tag{6.12}$$

  For detailed explanation, please refer to Appendix A.

- **In memory buffers:**

  Let M be the number of blocks that can fit in main memory at any one time. It should be noted that, M is the total number of blocks that can remain in the main memory for query processing purpose only. This does not include other buffers which may be there in the main memory for other purposes. Small values of M forces the same block to be read many times, thus increasing the cost of the join operations beyond what is apparently needed to read the input blocks and write the result.

In what follows, we will consider equijoins in detail. The same can be extended to range joins which may incur some additional decompression overheads. As was the case with selection, the cost depends on the presence or absence of indices on the join attribute and the type of index. In subsection 6.4.1, we will consider the possible ways to do the query processing when no index on the join attribute exists in any of the relations. Then, in subsection 6.4.2, will consider join processing when at least one index is present on the join attribute.

6.4.1 No Index on Join Attributes Exists

There are three widely used methods in this situation, namely **nested-loop join**, **sort-merge join** and **hash join**. We will discuss discuss these one by one.

- **Nested-Loop Join:**

  To estimate the cost of this operation, the way in which the product of two relations is computed must be considered first. It is assumed that neither of the two relations fit in the main memory. So the blocks must be shuffled in and out of the memory several times. Let us assume $B_{S_c} \leq B_{R_c}$ and $M \leq B_{S_c}$. The product is executed as follows:

  1) Divide S into segments of $M - 1$ blocks each.

  2) Read the first segment of S and in the remaining block in memory read each block of R, in turn, and 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.

If the join attributes are compressed using the same statistics, no decompression is required for matching. Otherwise, the corresponding attributes of both the relations have to be decompressed to do the comparison. During this operation, 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 $B_{S_c}/(M-1)$, the cost of the input of this operation is,

$$B_{R_c}B_{S_c}/(M-1) + B_{S_c} \tag{6.13}$$

Including the output cost given by equation 1.10, the total cost is,

$$B_{R_c}B_{S_c}/(M-1) + B_{S_c} + (B_{R_c}T_S + B_{S_c}T_R)/I_B \tag{6.14}$$

The benefit of compression can be best illustrated with the example relations R and S. Assuming typical values,$M = 101$, $I_B = 10000$. Then,

total cost for the uncompressed relations = 652500 disk accesses

total cost for the compressed relations = 202500 disk accesses

i.e., a savings of about 70% in disk accesses.

- **Sort-Merge Join:**

  The cost incurred is the sum of :

  1) The cost of sorting the relations, R and S.

  2) The cost of the join process.

  3) The output cost.

  Each pass of the multiway merge-sort, reads each block of the relation once. It also writes the new blocks that hold the intermediate result of the sort. Thus, in each pass 2B disk accesses are needed, if the relation is stored packed in B blocks. The number of passes required to sort the relation is $Log_M B$. Therefore, the cost of performing a multiway merge-sort is $2BLog_M B$. During the final join process, each block of the sorted relations have to be read once. For detail of merge-sort please refer [AHU83]. Finally, there is the output cost as given by equation 1.10. So the total cost of sort join is

$$2B_{R_c}Log_M B_{R_c} + 2B_{S_c}Log_M B_{S_c} + B_{R_c} + B_{S_c} + (B_{R_c}T_S + T_R B_{S_c})/I_B \qquad (6.15)$$

The first two terms are the cost of sorting the relations, the next two are the cost of scanning the sorted relations for matching pair of tuples, and the last term is the output cost. If the join attribute in the two relations are compressed using the same statistics, then the sort can be done on the compressed values of the attribute and the matching can be done without any decompression. Otherwise, the join attributes of the two relations have to be decompressed before sorting. The cost savings due to compression can be demonstrated with the example relations R and S. Assuming, $M = 101$ and $I_B = 10000$, as before,
total cost for the uncompressed relations = 223500 disk accesses
total cost for the compressed relations = 109500 disk accesses
i.e., a 51% savings in disk I/O.

- **Hash Join:**
In hash join, the relations to be joined, R and S, are *partitioned* into disjoint subsets and then the corresponding subsets are joined to produce the result. The method of partitioning is to first choose a hash function h, and a partition of values of h into, say, H_1, ..., H_n. Then, relation R is partitioned into disjoint subsets R_1, ..., R_n, where a tuple r of R is in R_i whenever $h(r)$ is in H_i. Here by $h(r)$ we mean the hash function applied to the joining attribute of r. Similarly, S is partitioned into corresponding subsets S_1, ..., S_n with s in S_i when $h(s)$ is in H_i. The subsets R_i and S_i are actually buckets. If tuple r in R is in R_i and it joins with a tuple s from S, then the joining attributes of r and s are equal, thus $h(r) = h(s)$ and s is in S_i. This is why, to join R and S, it suffices to join the subsets R_i and S_i for each i. The method proceeds as follows:
A hash function that hashes tuples of R and S into P buckets numbered 0 to $P - 1$

is selected. If $P > M$, R cannot be partitioned into all P buckets at once, because too many block accesses has to be done to swap blocks for the buckets in and out of main memory. Instead, R is divided into M "superbuckets", each representing P/M of the buckets. Only one block for each superbucket is kept in the main memory, amd when it is full, it is moved to the disk. A new block for that superbucket is then used in main memory. In the first pass, superbucket 0 represents buckets 0 through $(P/M) - 1$, superbucket 1 represents buckets P/M through $(2P/M) - 1$, and so on. Each block of the relation is read, each tuple is hashed, and if a tuple belongs to bucket i, it is placed in superbucket $\lceil iM/P \rceil$.

In general, the p^{th} pass, $p > 1$, starts with superbuckets each of which represents P/M^{p-1} buckets and ends with more, smaller superbuckets, each representing P/M^p buckets. During the pass, one superbucket is processed at a time, partitioning the buckets it represents into M groups of buckets; each group corresponding to one of the new superbucket. Each superbucket is thus, read once during the pass. M blocks representing the M smaller superbuckets are kept in main memory and are written to disk as and when they get filled. After $Log_M P$ passes, the new superbuckets represent single buckets and the partitioning is complete.

Each pass reads and writes about B blocks, if the relation is stored in B blocks. This is because, all the superbuckets into which the relation is partitioned is read, and the new superbuckets, with a finer partition, are written back. Thus, the cost of $Log_M P$ passes is $2BLog_M P$ disk accesses. So, the total cost of partitioning the compressed relations R and S is $2B_{R_c} Log_M P + 2B_{S_c} Log_M P$.

After the two relations are partitioned, the following steps are repeated for each subset R_i, $1 \leq i \leq P$, in the partition for R and its corresponding subset S_i in the partition for S.

- Read R_i and built a hash table for it in the main memory. It is assumed that a hash table of R_i fits into the main memory.

- Hash each tuple of S_i, with the same hash function as that in the above step. Probe for a match. If there is one, output the result tuple, otherwise proceed with the next tuple of S_i.

For these two steps, all the blocks of the compressed relation R and S is accessed once, resulting in $B_{R_c} + B_{S_c}$ disk accesses. Finally, there is the output cost of the join, as usual. No decompression is required since hashing and comparison are done on compressed values of the join attribute. Therefore, the total cost of the hash join is

$$2B_{R_c}Log_M P + 2B_{S_c}Log_M P + B_{R_c} + B_{S_c} + (B_{R_c}T_S + T_R B_{S_c})/I_B \tag{6.16}$$

The benefit of compression can be illustrated with the same example as in the previous cases, with $M = 101$ and $I_B = 10000$. We assume $P = 1000$, i.e., the number of disjoint subsets into which the relations are partitioned is 1000. P is taken to be large enough so as to ensure that the corresponding subsets of the two relations are small enough to fit in the main memory during the join phase. Then
total cost for the uncompressed relations = 210000 disk accesses
total cost for the compressed relations = 105000 disk accesses
a 50% savings in disk accesses.

6.4.2 Index on Join Attribute Exists

The following cases are possible:

1. **One clustering index on B exists:**
   We want to compute the join $R(A,B) \bowtie S(B,C)$ and let us assume that S has a clustering index on B. Then the join can be done 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, $1/I_S$ of the total tuples of S have to be fetched which in turn implies B_{S_c}/I_S block accesses. This results in $T_R B_{S_c}/I_S$ block accesses. Apart from this each block of R has to be read once. And finally, there is the output cost as usual. So the total cost comes to

$$B_{R_c} + T_R B_{S_c}/I_S + (B_{R_c} T_S + T_R B_{S_c})/I_B \tag{6.17}$$

Again, the savings in I/O can be illustrated with relations R and S. Let $I_B =$ 10000.Then,

the total cost for the uncompressed relations=210000 disk accesses

the total cost for the compressed relations=105000+(d_cost) disk accesses

The decompression cost *d_cost*, is about 20 disk accesses for integer or floating point attributes and about 100 disk accesses for character string attributes for relation R, as shown earlier. So, the decompression cost is by far offset by the reduction in number of disk accesses and an overall disk access savings of around 80% is achieved.

2. **One non_clustering index on B exists:**

   The only change in this case compared to clustering index is, the tuples to be accessed via the index can be located anywhere in the relation. So for each tuple in R, min(B_{S_c},$T_R T_S/I_S$) block accesses are required instead of $T_R B_{S_c}/I_S$. So the total cost stands at

$$B_{R_c} + T_R min(B_{S_c}, T_S/I_S) + (B_{R_c} T_S + T_R B_{S_c})/I_B \tag{6.18}$$

For the relations R and S, let $I_B = 10000$. Then,

the total cost for the uncompressed relations=1160000 disk accesses

the total cost for the compressed relations=1080000+(d_cost) disk accesses

Again, the decompression cost *d_cost*, is around 20 or 100 disk accesses depending on whether the join attribute is integer/floating point or character string for relation R. Although the percentage savings is only around 7 percent, the number of disk accesses saved is 80000 which is very significant. So, it easily offsets the additional cost incurred for decompression.

3. **Two clustering index on the join attribute, B, exists:**

   We assume that both S and R have a clustering index on B. So for each B value, the corresponding tuples from both the relations are accessed via the indices. So, for a particular B value the number of blocks accesses for relation R is max(1,B_{R_c}/I_R), and that for S is max(1,B_{S_c}/I_S). This has to be done for all B values. So the total cost is

$$I_B[max(1, B_{R_c}/I_R) + max(1, B_{S_c}/I_S)] + (B_{R_c}T_S + T_R B_{S_c})/I_B \tag{6.19}$$

   So for the relations R and S, with $I_B = I_R = I_S = 10000$,
   total cost for the uncompressed relations = 170000 disk accesses
   total cost for the compressed relation = 95000 disk accesses
   i.e., a 44% savings in disk access. No decompression is required because no comparison is necessary as the tuples are accessed via indices.

4. **Two non_clustering indices on the join attribute, B, exists:**

   All other things are the same as in the previous case (two clustering indices), except for the fact that now min(B_{R_c},T_R/I_R) blocks of R and min(B_{S_c},T_S/I_S) blocks of S has to be retrieved for each B value on an average. So the total cost is

$$I_B[min(B_{R_c}, T_R/I_R) + min(B_{S_c}, T_S/I_S)] + (B_{R_c}T_S + T_R B_{S_c})/I_B \tag{6.20}$$

   Since the first term in the above quantity, the input cost, does not depend on compression. But since the output cost of joins is significant, and the output cost depends on compression, performance improvement is still achieved. For relations R and S, with $I_B = I_R = I_S = 10000$,
   total cost for the uncompressed relation = 350000 disk accesses
   total cost for the compressed relations = 275000 disk accesses
   i.e., a 21.4% savings in disk I/O and no decompression is required.

5. **One Clustering index and one non_clustering index:**

   We assume that S has a clustering index and R has a non_clustering index on B. Then for each B value, the number of blocks of S retrieved is $max(1, B_{S_c}/I_S)$ and the number of blocks of R retrieved is min(B_{S_c},T_R/I_R) on an average. So the total cost is

$$I_B[max(1, B_{S_c}/I_S) + min(B_{S_c}, T_R/I_R)] + (B_{R_c}T_S + T_R B_{S_c})/I_B \qquad (6.21)$$

   We illustrate the benefits of compression with the usual example. Let $I_B = I_R = I_S = 10000$. Then,

   total cost for the uncompressed relations = 260000 disk accesses

   total cost for the compressed relations = 185000 disk accesses

   i.e., a 28.8% savings in disk I/O and no decompression is required.

We have thus demonstrated that database compression can significantly improve the performance of all types of joins – which are by far the most expensive of all the relational operations.

6.5 Update Operators

The update operators are the ones that change the contents of the database. These operators are **insert**, **delete** and **update**. The implication of database compression on these operators will be discussed in this section.

Before proceeding farther, the way to locate a page having a desired amount of **free space**, in a specified relation, will be considered. In order to facilitate this, each page in a relation is stored packed. In other words, there is no intervening free space between any two records in a page. All the free space in a page occurs after the last record in the page and before the beginning of the attribute pointer information. This is done because the compressed records are of variable size. So, unequal sized holes in between successive records, resulting from deletions, may not be filled by subsequent inserts. This leads to the problem of **fragmentation** in a page. Once the pages are maintained with all the free space appearing after the data records, a simple data structure, called **free space list** is used to find appropriate pages. The data structure consists of a number of lists. Each list contains several page numbers, indicating that each of the individual pages in the list has free space lying within a certain range, the range being determined by the list under consideration. For example, a range of 50 bytes can be alloted for each list, i.e., all pages on list 1 have free space in the range of 0 to 50 bytes, those on list 2 have free space in the range of greater than 50 to 100 bytes and so on. The choice of the range that is alloted to each list is guided by the average size of compressed records. Intuitively speaking, the range can be chosen to be slightly greater than the average size of the compressed records. This ensures that one more compressed record can possibly fit into pages on list 1, 1-2 more compressed records can possibly fit into pages on list 2 and so on. In the above example, where 50 bytes is alloted to each list, if we want to insert two compressed records, one of size 50 bytes and the other of size 60 bytes, a page from list 4 having free space in the range 150-200 bytes can be chosen, and the records can be placed in the selected page. This will reduce the amount of free space available in that page and it should be moved to the appropriate list. Thus, a sequential scan of the pages of the relation for locating pages having sufficient free space is averted. Given the amount of free space required, one disk access is required to find an appropriate page. This is because all that has to be done is to fetch one block from the appropriate list to main memory and read a page number from that block.

6.5.1 Insert

The insert operation involves the insertion of one or more records into a given relation. The steps involved in inserting a record are:

- the record to be inserted is compressed.
- a page having sufficient free space to hold the compressed record and the attribute size information is selected by the method described above.
- the page is accessed and the record is inserted.

- the modified page is written back to the disk.

Therefore, the cost of inserting a record into a compressed relation is that of accessing one block from *free space* list to find an appropriate page plus that of accessing that page and writing that page back after insertion. So,
total cost = 3 disk accesses
This is at most one disk access more than that for the corresponding uncompressed relation. If more than one record, say, two or three records, are to be inserted, then a page having sufficient free space can be selected and all the records can be inserted in that page. This keeps the number of disk accesses to 3, as in the case of a single record. However, if a large number of records are to be inserted, then they are usually appended to the end of the relation by allocating sufficient number of new pages. If n_i be the number of records to be inserted into the compressed relation R, the number of new pages that must be allocated is around $n_i B_{R_c}/T_R$. This implies a cost of

$$n_i B_{R_c}/T_R \tag{6.22}$$

So, for example, if $n_i = 10000$, then
total cost for the uncompressed relation = 1000 disk accesses
total cost for the compressed relation = 500 disk accesses
a 50% savings in disk accesses. So, while there is practically no additional penalty for inserting few records, there is substantial savings in disk accesses for bulk insertion if the relation is kept compressed.

6.5.2 Delete

The delete operation involves the deletion of one or more records from a relation. The following steps are involved in deleting a record:

- the record to be deleted is selected.

- the record is removed by overwriting it with the record following it in the page, i.e., the next record is moved up by an amount equal to the size of the deleted record .

Similarly, all subsequent records are moved up by the same amount. This ensures that no holes remain in between successive records.

- the modified page is written back to the disk.

So, the cost of deleting a number of records is the cost of selecting the records plus the cost writing back the modified pages to disk after deletion. If, however, all the records in a page are deleted, the page need not be written back. The cost of selection depends on the presence or absence of index on the selection attribute for deletion and has been discussed in detail in section 6.2. The additional cost here, it that of writing the modified pages back to the disk. The number of such pages depends on the number of tuples modified and their location in the relation. Assuming, that *selectivity* is s for the query under consideration, $T_R s$ tuples has to be deleted. Note, $s = 1/I$, if the query is of exact match type. Now, in the worst possible case, all the $T_R s$ tuples can be located on a different page, resulting in $T_R s$ disk accesses. If, $T_R s \geq B_{r_c}$, the maximum number of disk accesses needed can only be B_{R_c}. The two cases can be combined to say that the number of disk accesses required is $min(B_{R_c}, T_R s)$. However, if the relation is ordered on the attribute on which selection for deletion is done, then the tuples to be deleted will be located on $B_{R_c} s$ consecutive blocks. Out of these consecutive blocks, at most the first and the last page may need to be written back as they may still contain some tuples which are not deleted. For all the pages inbetween, all the records are deleted, eliminating the need to write them back. So, the additional cost, apart from selection, is 2 disk accesses at most. To sum up, the total cost is

$$SC + 2 \tag{6.23}$$

when the relation is ordered on the selection attribute.

$$SC + min(B_{R_c}, T_R s) \tag{6.24}$$

otherwise. Note that SC is the selection cost.

We have already shown in section 6.2, that significant savings in selection cost results if the relation is kept compressed. This will be reflected in the first term of the above two equations. Moreover, the second term of the second equation, that is $min(B_{R_c}, T_R s)$, will also benefit from compression if $B_{R_c} \leq T_R s$. This will result is farther reduction in disk accesses as compared to the case when the relation is kept uncompressed and an overall savings of 50% can be expected for 50% compression.

6.5.3 Update

The update operation modifies one or more records in a relation. The steps involved in updating a record are as follows:

- the record to be modified is selected.
- the record is modified. This involves decompression of the attribute(s) to be modified, modification of the appropriate attribute(s), compression of the modified attribute(s) and concatenation of the modified compressed attribute(s) with the unchanged compressed attributes to form the updated compressed record.
- re-insertion of the updated record to a possibly new page. This is because the size of the updated compressed record, in general, is different from that of the original compressed record. So, this step basically involves the deletion of the record followed by its insertion.

So, the cost of an update is the cost of selection of the records to be updated plus the write-back cost, which involves the cost of writing back the blocks that originally contained the tuples that are updated, and the blocks into which the updated tuples may have been re-inserted. The cost of selections has already been explored in section 6.2. As in deletion, the number of tuples to be updated is T_R/s and $s = 1/I$ for exact match queries. Now, in the worst case, each of these tuples can be located on a separate page and their retrieval may require upto $min(B_{R_c}, T_R s)$ block accesses. Assuming the worst possible scenario, that each updated tuple has to be relocated, three block accesses are required for each of them (see Insert), resulting in $3min(BR_c, T_R s)$ block accesses. Moreover, the original blocks that contained the tuples updated have been modified in the process. So, they have to be written back as well, resulting in $min(B_{R_c}, T_R s)$ more disk accesses. Therefore, the maximum upper bound on the total cost is

$$SC + 4min(B_{R_c}, T_R s) \tag{6.25}$$

where SC is the selection cost. For uncompressed relations having fixed-size records, the second term is absent because update can be done *in place*. Therefore, although the first term results in significant savings for compressed relations(upto 50%), the overhead caused by the second term offsets its effect to produce some additional overhead compared to the corresponding uncompressed relations. However, if the records in the uncompressed relations are themselves of variable size, the second term must be included for the cost calculation for the uncompressed relations as well. So, on compression upto 50% savings in disk accesses can be expected. The situation is slightly different when the relation is ordered on the attribute on which the selection is done. Then, the $T_R s$ tuples to be updated are located on approximately $B_{R_c} s$ consecutive pages. All the records in these pages are updated. It is assumed that although the sizes of individual records change on compression, the average size of the compressed records remain more or less the same. Therefore, we can expect the modified records to fit in the same $B_{R_c} s$ pages in which they were stored before update. So, only $B_{R_c} s$ blocks have to be written back, resulting in a total cost of

$$SC + B_{R_c} s \tag{6.26}$$

A selection cost savings of about 50% occurs if the relation is kept compressed as shown in section 6.2. The second term, which is directly proportional to the number of blocks on which the relation is stored, also result in 50% savings, assuming 50% compression, as we have done throughout this analysis. So, an overall savings of upto 50% can be achieved in this case.

6.6 Conclusion

In this chapter, it has been shown that database compression can result in significant reductions in disk accesses. Today, even PCs are equipped with powerful processors. So the decompression required in certain cases is not a serious bottleneck. This can be seen from the percentage savings achieved in terms of the number of disk accesses for the example relations R and S. The results for the various relational operators are summarized in Table 6.1. Although, in the case of joins, the percentage savings is not always very high, the actual savings in the number of disk accesses is very large. This is because joins involve a large number of disk accesses and even a small percentage savings implies a significant reduction in the number of disk accesses.

From the preceding discussions, it is not very difficult to see the profound impact of database compression on the response time of transactions, particularly in the multiuser environment. Firstly, the response time of the individual transactions decrease because of reduced I/O requirements. Secondly, the blocking time of transactions due to concurrency requirements decrease. This is because individual transactions take shorter time to read,write and commit, thereby reducing the waiting time of the transactions blocked on them. Therefore, database compression, if smoothly integrated with other performance enhancement techniques like indexing, concurrency control, disk scheduling etc., will enable the database designer to maximize the performance attainable from the available resources.

Table 6.1: Performance Summary

Relational Operator	**Index Type**	**% Savings** (*disk accesses*)	Formula (equation no.)
1. **Select**			
a. Exact Match	Clustering	upto 50	(6.3)
	Non-clustering	50, if $B_{R_c} \leq T_R/I$ 0, if $B_{R_c} > T_R/I$	(6.4)
	No index	50	(6.5)
b. Range	Clustering	50	(6.6)
	Non-clustering	50, if $B_{R_c} \leq T_R s$ 0, if $B_{R_c} > T_R s$	(6.7)
	No-index	50	(6.8)
2. **Project**	—	50	(6.9)
3. **Join**			
a. Without index			
* Nested-loop	No index	70	(6.14)
* Sort-Merge	No index	51	(6.15)
* Hash join	No index	50	(6.16)
b. One index	Clustering	80	(6.17)
	Non-clustering	7	(6.18)
c. Two indices	Both Clustering	44	(6.19)
	Both Non-clustering	21	(6.20)
	One clustering, other non-clustering	29	(6.21)
4. **Updates**			
a. Insert	—	0, for single record 50, for bulk insert	(6.22)
b. Delete	Clustering	upto 50	(6.23)
	Non-clustering	upto 50	(6.24)
	No index	upto 50	(6.24)
c. Update	Clustering	upto 50	(6.26)
	Non-clustering	upto 50	(6.25)
	No index	upto 50	(6.25)

Appendix A

Output Cost of Joins

Let $R(A,B) \bowtie S(B,C)$ represent the join of two relations R and S, and suppose that for the join attribute B, either the domain of B in R is a subset of the domain of B in S, or vice versa. Assume that each tuple in the product of the domains of all the attributes of R is equally likely to be in R, and similarly in S. Then the expected size of $R(A,B) \bowtie S(B,C)$ is

$$T_R * T_S / I_B \tag{A.1}$$

where I_B is the size of the domain of B, in R os S, whichever is the superset of the other.

To show the above result for the join $AB \bowtie BC$, the following assumptions are made:

1. there is an inclusion dependency on attribute B; specifically, every B-value in the relation AB also appears in the relation BC, but not necessarily vice versa.

2. let D_A, D_B, and D_C be the domains of A in AB, B in BC and C in BC respectively. Let E_B be the domain of B in AB. Note, $E_B \subset D_B$ from assumption 1.

3. let I_A , I_B and I_C be the distinct values of attributes A in AB, B in BC and C in BC respectively. Let J_B be the same for B in AB. Note, I_A, I_B, I_C and J_B are the sizes of D_A, D_B, D_C and E_B respectively.

4. let T_{AB} be the number of tuples in AB and T_{BC} be that in BC.

To calculate the size of $AB \bowtie BC$, consider an arbitrary tuple abc in $D_A * D_B * D_C$.

The number of such tuples is $I_A I_B I_C$. For abc to be in the join, ab must be in AB and bc must be in BC. By assumption 5, the probability that a random tuple bc is in BC is $T_{BC}/I_B I_C$, i.e., the number of tuples in BC divided by the number of tuples in the product of domains $D_B * D_C$. Now, the probability that ab is in AB must be found. If b is a value in D_B and not in E_B, then this probability is zero. This case occurs $(I_B - J_B)/I_B$ of the time. However, J_B/I_B of the time, b will be in E_B. Thus the probability that a random tuple chosen from $D_A * D_B$ is in AB is

$$(I_B - J_B) * 0 + (J_B/I_B)(T_{AB}/I_A J_B) = T_{AB}/I_A I_B \qquad \text{(A.2)}$$

Now, the probability that ab is in AB is multiplied with the probability that bc is in BC to obtain the probability that abc is in $AB \bowtie BC$; it is

$$T_{AB} T_{BC}/I_A I_B^2 I_C \qquad \text{(A.3)}$$

Finally, this probability is multiplied by the number of possible tuples, which is $I_A I_B I_C$, to get the estimated size of $AB \bowtie BC$, which is

$$T_{AB} T_{BC}/I_B \qquad \text{(A.4)}$$

Bibliography

[Huf52] D. A. Huffman. "A Method for Construction of Minimum-Redundancy Codes," *Proceedings of the IRE*, 40: pp:1098-1101 (September 1952).

[Rth72] S. S. Ruth, and P. J. Krentzer. "Data Compression for Large Business Files," *Datamation*, pp:62-66 (September 1972).

[Reg81] H. K. Reghbati. "An Overview of Compression Techniques," *IEEE Computer*, pp:71-75 (April 1981).

[Sev83] G. D. Severance. "A Practitioner's Guide to Database Compression – A Tutorial," *Information Systems* 8(1): pp: 51-62 (January 1983).

[AHU83] A. Aho, J. E. Hopcroft and J. D. Ullman. "Data Structures and Algorithm," Addison-Wesley, Reading Mass (1983).

[Wel84] T. A. Welch. "A technique for High Performance Data Compression," *IEEE Computer*, pp:8-19 (June 1984).

[Knu85] D. E. Kunth. "Dynamic Huffman Coding," *Journal of Algorithms*, 6, pp:163-180 (1985)

[Bas85] M. A. Bassiouni. "Data Compression in Scientific and Statistical Databases," *IEEE Transactions on Software Eng.*, SE-11(10): pp:1047-1048 (October 1985).

[Cor85] G. V. Cormack. "Data Compression in Database Systems," *Communications of the ACM*, pp:1336-1350 (December 1985).

[Sha86] L. D. Shapiro. "Join Processing in Database Systems with Large Main Memories," *ACM Transactions on Database Systems*, Vol.11, No.3, pp:239-264 (September 1986)

[Wit87] I. H. Witten, et al. "Arithmetic Coding For Data Compression," *Communications of the ACM*, pp:520-540 (June 1987).

[CL88] M. J. Carey and M. Livny. "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication," *Proceedings of* 14^{th} *VLDB Conference*, Los Angeles, California, (1988).

[Jon88] D. W. Jones. "Application of Splay Trees to Data Compression," *Communications of the ACM*, 31(8): pp:996-1007 (August 1988).

[Ulm89] J. D. Ullman. "Database and Knowledge-base Systems," Vol 2., Computer Science Press.

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

[RH93] M. A. Roth and S. J. Van Horn. "Database Compression," *SIGMOD Record*, Vol. 22, No. 3: pp:31-39 (Sept. 1993)

[Grf93] G. Graefe. "Options in Physical Database," *SIGMOD Record*, Vol. 22, No. 3, Sept. 1993.

[BW94] B. R. Iyer and D. Wilhite. "Data Compression Support in Databases," *Proceedings of the VLDB Conf.*, Santiago, Chile, 1994.