Title
A NEW COMPRESSION METHOD WITH FAST SEARCHING ON LARGE DATABASES

Authors
Li, J.Z.
Rotem, D.
Wong, H.K.T.

Publication Date
1987-03-01

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA, BERKELEY

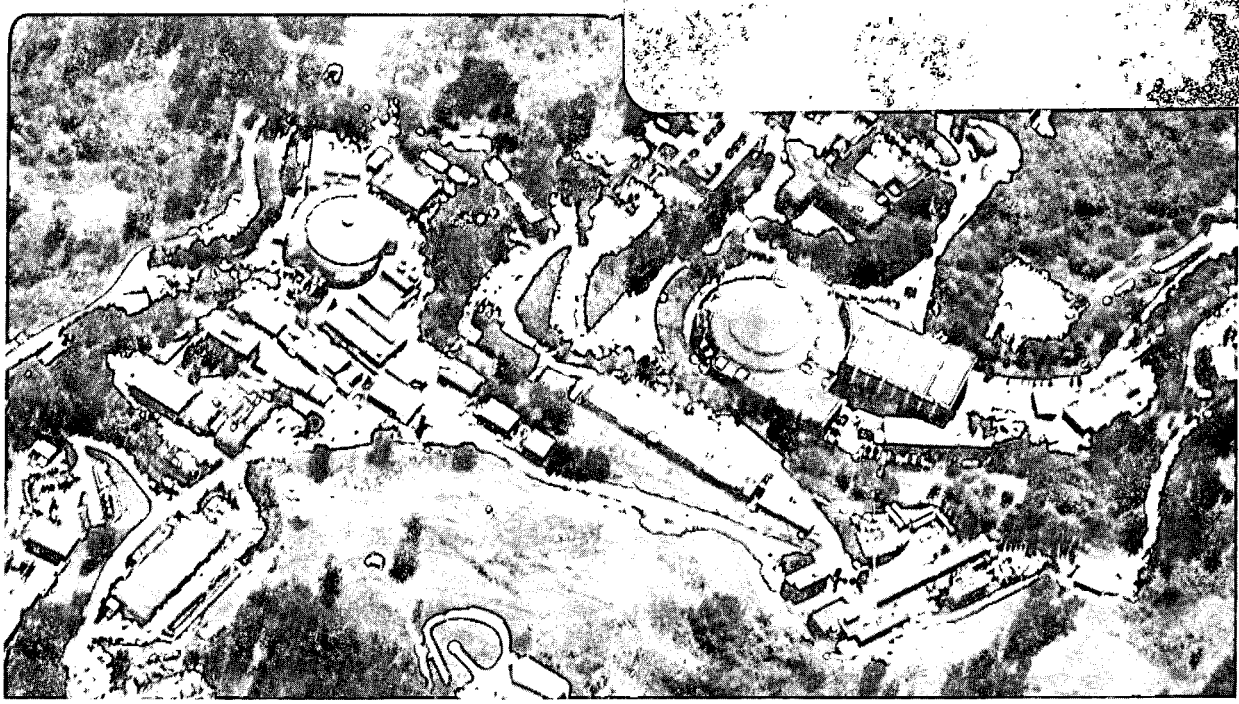
Information and Computing Sciences Division

To be presented at the 13th International Conference on Very Large Data Bases, Brighton, England, September 1–4, 1987

A NEW COMPRESSION METHOD WITH FAST SEARCHING ON LARGE DATABASES

J.Z. Li, D. Rotem, and H.K.T. Wong

March 1987



Prepared for the U.S. Department of Energy under Contract DE-AC03-76SF00098

DISCLAIMER

LBL-22393

A New Compression Method with Fast Searching on Large Databases

J.Z. Li, D. Rotem, and H.K.T. Wong

Computer Science Research Department
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

March, 1987

This research was supported by the Applied Mathematics Sciences Research Program of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098.

A New Compression Method with Fast Searching on Large Databases

Jian-zhong Li*, Doron Rotem** and Harry K. T. Wong***

Lawrence Berkeley Laboratory,
University of California
Berkeley, California 94720

Abstract

In this paper, a new compression method for constant removal from very large scientific and statistical databases is presented. The new method combines the best features from several classical constant removal compression methods. The result, both analytical and experimental, shows that the method is superior to these popular methods in terms of compression effectiveness and efficient searching on the compressed data. In addition to the development, analysis and validation of this new method, this paper also presents analysis of several traditional constant removal methods for the purpose of analytic comparison. A large collection of experiments have been designed and run to observe and validate the behavior of the compression methods. Another contribution of the paper is that performance characteristics are identified for different compression methods under different data properties assumptions. The result can be used as a basis of selecting compression methods by matching the properties of the database at hand to the data properties experimented in the paper.

Supported by the Office of Energy Research, U.S. DOE under Contract No. DE-AC03-76SF00098.

* on leave from Dept. of Computer Science, Heilongjiang Univ., China.

** on leave from University of Waterloo, Canada.

*** Now with Ashton-Tate Advanced Development Center.

1. Overview and Motivation

We are interested in very large Scientific and Statistical databases (SSDBs) ([5], [4]). SSDBs are prevalent in scientific, socio-economic, and business applications. Examples of SSDBs are experiments and simulations for scientific applications, census, health, and environmental data for socio-economic applications, and inventory and transaction analysis for business applications. These databases are often very large and sparse. Examples are scientific databases such as earthquake monitoring databases and high energy physics databases where the so-called noise (values that fall below certain threshold values) are converted into nulls or zeros (referred to as *constants* in the literature). The volume of these data is large because they are typically collected by automatic devices and most of the collected data are noise because interesting events typically scatter between long periods of inactivity.

Other kinds of databases such as summary databases prevalent in SSDBs are also very large and sparse. An example is the summary data of a multi-factor parametric experiment of corrosion of materials under different conditions such as temperature, acidity, salinity, and duration. Attributes such as material, temperature, acidity, salinity, and duration represent parameter data. Attributes such as amount of corrosion is the measured data. The former kind of attributes are referred to as *category attributes*, and the latter is referred to as a *summary attribute*. Two factors cause these SSDBs to be extremely large. First, they may contain hundreds of summary attributes. Second, the cardinalities of the category attributes can themselves be quite large; and the number of tuples generated is the product of these cardinalities. These databases can be quite sparse. In the corrosion

experiment database, suppose that temperature does not have too much effect on certain types of material, then in the corrosion column there would be the same value in consecutive positions for all the acidity, salinity, and time.

To remove the large collection of constants is an essential requirement for managing these SSDBs. This requirement rules out a large body of compression techniques in the literature that deals with transmission and text compression such as Huffman encoding [8] and those in [1,7,8,9,10,11,12] as mentioned in [13].

In addition to the amount of compression one can achieve using a compression method, there are two other important requirements that we stress in evaluating and selecting a compression method. The first is the ability to perform efficient and random searching in the compressed database, given a logical position of the original database. This requirement is essential to provide a querying capability on the database using the compressed data structure as the physical structure. Another equally important requirement on a compression method is the capability to provide an efficient mapping from arbitrary positions in the compressed data back to the corresponding logical positions in the original database.

The following example illustrates the importance of this requirement. Because of the size and sparsity of SSDBs as mentioned before, it is desirable to have operators that can directly operate on compressed data without first decompressing it. Operators such as transposition and aggregation are good examples of why direct manipulation on compressed data is desirable [17]. These operators, however, rely heavily on the ability to discover efficiently, given an arbitrary position in the compressed data, the corresponding logical position in the original database, in order to reposition the data items in the new transposed space. To continue our example of corrosion-experiment data, instead of ordering the data in the order of material, temperature, acidity, salinity, and duration, a transposition request to reorder the data so that now the data space is linearized in the order of, say, material, duration, salinity, acidity, and temperature. For each data item in the compressed data, a backward mapping is necessary to discover the coordinates of the original space, so that a new position can be computed corresponding to the new requested space. To have an efficient transposition algorithm depends strongly on the ability to perform fast backward mapping. Classical methods such as run-length encoding [6] and its derivatives such as header compression [2,3] provide good performance in terms of removing long runs of constants, but they have a poor forward and backward mapping capability. Also, these methods can not be used on dynamic database environment where additions and deletions may be required.

In this paper, a new constant removal compression method is proposed and is shown to be superior to the classical methods and their derivatives both analytically and experimentally in most situations. It also lends itself to dynamic changes in databases. There are three other important contributions in this paper. First, analysis in terms of compression ratios and searching efficiency is given for the new compression method as well as for three other competing constant removal methods. Second, a large collection of experiments has been performed to validate the analytic results of these methods. Third, data characteristics have been identified under which a compression method can be selected to provide the best compression ratio.

The rest of the paper is organized as follows, the next section provides some background and terminology for the sections to follow. Section three discusses our new compression method, called BAP. Experiments and analysis results are given in Section four. In that section, comparison of BAP with other constant removal methods is given to provide a performance metric under different situations. Finally, Section five provides some conclusion and a discussion for future work. An appendix with a list of most the commonly used symbols in the paper is also provided.

2. Background

In this section, some important terms on constant removal compression will be introduced. Also, we will survey three popular techniques for constant removal. They are bit map, run-length encoding and header compression. The terms *logical database* and *physical database* are used to refer to the uncompressed

and compressed database respectively. The *forward mapping* is a mechanism that determines the position in the physical database for a given position of a value in the logical database. The *backward mapping* is a mechanism that determines the position in the logical database for a given physical position in the physical database.

2.1. Bit Map

A bit map compression scheme consists of a bit map and a physical database which stores the non-constant values. The bit map is employed to indicate the presence or absence of non-constant data. The following example shows how the bit map compression scheme can be employed to implement a version of constant suppression.

Example:

Original data string

d1, c, c, d2, c, c, c, d3.

Compressed data string

bit map: 10010001.

physical database : d1, d2, d3.

For the bit map compression method, the mapping mechanism must search the whole bit map for both forward and backward mapping. And thus, the access time for both forward and backward mapping is O(N), where N is the number of bits in the bit map or equivalently the number of elements in the database.

2.2. Run-Length Encoding

The application of run-length encoding to constant removal is that each consecutive run of constants (there can be a few different types of constants to be removed) is replaced by a triple consisting of a separator SEP, an encoding of the constant X to be removed and a counter C indicating the length of the run. Of course the decoding algorithm must be given sufficient information which enables it to correctly interpret each component of a triple as well as additional values which may be present in the data stream without any compression.

To search a run-length encoded database for both forward and backward mapping, a sequential search has to be done to sequence through the data, counting the number of unsuppressed values and references to the number of repeated values. The time required here is again O(N).

2.3. Header Compression

The header compression scheme is shown below. The vector L represents the uncompressed form of a database, in which the 0's are the constant to be suppressed and the V's are the unsuppressed values. Beneath the vector L is the list of counts which comprise the compression header, H. The odd-positioned counts hold accumulations of unsuppressed values; and the even-positioned counts hold the accumulations of zeros. The physical, compressed form of the data is represented by P.

L: V1 V2 0 0 0 0 0 0 0 0 0 V3 V4
V5 V6 V7 0 0 V8 V9 V10 0 0 0

H: 2,9,7,11,10,14

P: V1 V2 V3 V4 V5 V6 V7 V8 V9 V10

For the header compression method, both forward and backward mapping can be processed by binary searching on the header, H. The header may be organized as a B-tree or accommodated in fast storage if it is sufficiently small. Both forward and backward mappings require O(log s) time where s is the size of the header.

3. The BAP Compression Scheme

In this section we consider a new compression method which incorporates the advantages of several existing techniques.

The compression technique presented here constructs a physical database which consists of three parts: *Bit vector (BV)*, *Address vector (AV)*, and *Physical vector (PV)*, and is therefore called the BAP compression technique.

3.1. An overview of the BAP method

Let DB=(x_1,x_2, $\cdots$,x_N) be a logical database, and c be the constant to be suppressed. The bit vector, BV, indicates locations of constants and non-constants in the database and

will be stored on disk in a compressed form as explained later. The physical vector PV is the vector of the non-constants in DB, i.e.

$$PV = (y_1, y_2, \cdots, y_n)$$

where, y_j are in DB and $y_j \neq c$. The y_js are arranged according to their logical order in DB for $1 \leq j \leq n$, $n \leq N$ and are stored on disk. It is assumed that non-constants cannot be compressed, hence no compression algorithm is applied to PV. Finally, the address vector AV is typically small and will be used as an index for searching in the database; we will show that in most practical applications it can be stored in fast storage.

The idea behind our method is that in addition to efficient compression we need fast forward and backward mapping capabilities between the logical and physical database. In fact we will show that after compressing the logical database DB, we can find what is the position in PV of any of the x_i's in one disk access, provided that AV is maintained in fast storage. Conversely, given a position j in the physical database PV, we can determine what was its original location in DB, again using only a single disk access.

We now describe the components BV and AV in more detail.

3.2. Bit Vector (BV)

The bit vector is

$$BV = (b_1, b_2, \cdots, b_N)$$

where,

$$b_i = \begin{cases} 1 & \text{if } x_i \neq c \\ 0 & \text{if } x_i = c \end{cases}$$

Since BV is a bit vector of N bits, we can compress it using run length encoding in which we replace each run of zeroes by a counter which indicates the length of this run. There are several methods for achieving efficient run length encoding, the main problem here is how to encode the counters and provide a separator between counters. We chose to use in our case the Golomb encoding method [14,15] which was proven to achieve a compression ratio close to the information theoretic lower bound.

We had to modify this method so that we could also search in the compressed data. To this end, we divide BV into subvectors of D bits each, where D is a parameter chosen by the user. We will later show a mathematical analysis of how D is determined in order to maximize the efficiency of our compression scheme, subject to limitations such as storage size, block size and required response time. Each subvector in BV is compressed independently using Golomb's method and stored on a separate disk block (or a sequence of a few consecutive blocks). We now give an overview of Golomb's encoding method.

3.2.1. Golomb's encoding scheme

We define the compression ratio of a binary vector to be the ratio between the number of bits it occupies before and after it is compressed. The efficiency of Golomb's scheme is achieved by encoding the counters and the separators using a special method. The codeword for each counter is constructed as follows. A parameter m is chosen as explained later, and then each run of r consecutive zero bits is divided into $\lceil r/m \rceil$ groups consisting of m bits each except for the last group which may contain less than m bits. Each group is encoded by a 1-bit and the last group is encoded by a counter of fixed length. The idea of the method is illustrated below. The codeword for each counter consists of a variable-length *prefix* which has a 1-bit for each group of m 0-bits, and a fixed-length *tail* which counts the number of 0-bits in the last group. The prefix consists of $\lfloor r/m \rfloor$ 1-bits followed by a 0-bit as a separator. The tail consists of a binary number of $\lceil \log_2 m \rceil$ bits and its value is $r - m \cdot \lfloor r/m \rfloor$. It is shown in [15] that in order to maximize the efficiency of the method, the parameter m should be chosen as the integer such that p^m is as close as possible to 0.5, where p is the 0-bit probability. The compression ratio is shown in [15] to be a random variable with expected value R where

$$R = \frac{1}{(1-p)(\log_2 m + (1-p^m)^{-1})}$$

where $m = -\frac{1}{\log_2 p}$.

3.2.2. Using Golomb's method in BAP

Turning back now to the BAP method, we will compress each subvector in BV independently and store it on a disk block. Let us assume that the probability P(c), of finding a constant in each location of the database is fixed and independent of other locations. From [15], we know that the expected value of the compression ratio in this case is

$$R=\frac{1}{(1-P(c))(\log_2 m+(1-P(c)^m)^{-1})}.$$

where $m=-\frac{1}{\log_2 P(c)}$.

We are now in a position to explain how the user should choose the parameter D. It is intuitively clear that we are interested in having D as large as possible to maximize the compression efficiency. On the other hand as we explain later, in order to ensure that no more than k block accesses will be required for searching in the database, we require that each compressed subvector will fit on a sequence of at most k consecutive blocks. In most practical applications we will require k=1. Given a block size of S bits and an expected compression ratio R, it follows from the above restriction that the size of the subvector D should be chosen such that after compression each subvector of D bits should with a very high probability occupy less than kS bits. We found experimentally (see Table 1) that the distribution of the compression ratio has a very small standard deviation for all practical values of P(c) (typically less than 1.2 percent of R). Hence by using a compression ratio $R' = .98R$ which is slightly smaller than the expected compression ratio R, we ensure with extremely high probability that no overflow will occur when we choose D to be the maximum integer such that

$$\frac{D}{R'} \leq kS$$

In practice, a small overflow area may be assigned to the file so that in the unlikely event that any of the compressed subvectors requires more than kS bits, the remaining bits will be stored in the overflow area. Assuming no overflow, the compressed database will occupy $\lceil N/D \rceil$ blocks, each block of size S bits.

3.3. Address Vector (AV)

The division of BV into subvectors imposes a division of the database DB into $d=\lceil N/D \rceil$ sections, each consisting of D elements. In each one of these sections we may have zero or more non-constants. We define the address vector as

$$AV = (a_1, a_2, \cdots, a_d)$$

where, $a_1=0$, and for $i \geq 2$, a_i is the relative position in PV of the last non-constant element in the $(i-1)^{th}$ section of DB if such a non-constant exists; otherwise (all elements in the $(i-1)^{th}$ section are constants) , we set $a_i=a_{i-1}$.

The key point of this compression technique is that AV can reside in main memory by choosing the parameter D to be sufficiently large. Our experiments indicated that for all practical database sizes and values of P(c), no further compression of AV is needed. However, if the size of AV turns out to be larger than the available memory, we can take advantage of the fact that the difference between two successive elements in AV is very small (and bounded by 2D) as compared to the absolute value of each element, which depends on the size of PV.

In order to compress AV, we may use a *relative encoding* method [16] in which we store the difference of two consecutive elements instead of storing the actual elements. The method can be illustrated by the following example.

Example:

Original elements of AV

1000 1500 2000 2500 3000 ..... 10500 11120

Relative encoding

1000 500 500 500 500 700

In this way, the size of each element is encoded in $\log_2(2D)$ bits instead of $\log_2(|PV|)$ bits where $|PV|$ denotes the number of elements in the physical vector.

3.4. An Example of BAP

The following example will be used to illustrate the BAP compression technique. For simplicity we will not specify BV and AV in their compressed form.

Given
DB = (1,0,0,4,0,0,0,0,0,0,0,8,0,0,0,12,0,0,0,0,17,0, 0,20), let the constant be 0 and assume D = 5. Using BAP, the database DB will be compressed as follows

BV = (1,0,0,1,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,1,0,0,1)

PV = (1,4,8,12,17,20),

and

AV = (0,2,2,3,4).

Let us consider the element a_8 in AV. Since it is equal to a_2 we know that there are no non-constants in the second section of DB. As a further illustration, we note that the last non-constant in the third section of DB is 8 which appears in the third position of PV. We therefore have a_4=3.

3.5. Forward Mapping

In this section we discuss the mapping from the logical database to the physical database.

Given the ordinal position, LP, of a desired instance in the logical database, we want to determine whether this instance is a suppressed constant or an unsuppressed value. In the case it is unsuppressed, we would like to find its position in the physical vector (PV). In the description of the mappings, we assume k=1, i.e., each subvector is compressed into a single block. A generalization to any k is straightforward. The algorithm is as follows.

ALGORITHM FM

SN:= ⌈LP/D ⌉; offset:=LP mod D;

READ block with relative position SN in compressed BV

and decompress it into a buffer.

IF bv(offset)=0 **THEN** return DB(LP)=c

ELSE return DB(LP)=PV(AV(SN)+bitsum(offset));

where, the function bitsum(x) is a function which counts the number of 1's among the first x bits in the buffer and bv(j) is a function which gives the value of the j^{th} bit in the buffer.

Consider the following example where D=5,

BV = (1,0,0,1,0,0,0,1,0,0,0,1,0,0,0,0,1,0,0,1)

PV = (2,5,9,13,18,21),

and

AV = (0,2,3,4).

Let the required logical position be LP=12. Then ⌈LP/D ⌉= ⌈12/5 ⌉=3, we read in the third subvector of BV into the buffer, (in this example no decompression is performed) and find that bv(2) is a 1-bit and bitsum(2)=1. Thus,
DB(12)=PV(AV(3)+1)=PV(3+1)=PV(4)=13.

THEOREM 1. Given that AV is in fast storage, Algorithm FM requires one disk access to find the position in PV corresponding to a logical position LP and two disk accesses to find the value of the element in logical position LP of DB.

Proof. From the algorithm FM, it is obvious that since AV resides in main memory, we need to access a single block of BV in order to find the position in PV of the element which is in position LP in DB. In the case that the required element is a non-constant and we need its actual value, one more block access in PV is needed. All the other operations specified by the algorithm do not require access to the disk Q.E.D.

3.6. Backward Mapping

Given a position PP of a non-constant element y in PV, we want to find the logical position LP of this element in DB. The algorithm is a little more complicated than the forward mapping algorithm. As before, we denote the number of elements in AV by d where d= ⌈N/D ⌉.

ALGORITHM BM

(1) We perform a binary search in AV to find the first entry larger than PP.

$I:=\min\left\{ i \mid PP \le AV(i) \right\}$ if $PP \le AV(d)$, otherwise I:=d+1;

Comment: At this point we know that the required non-constant y is in the $(I-1)^{st}$

section of DB;

(2) **READ** the $(I-1)^{st}$ block of BV and decompress it into the buffer;

(3) set j:=PP–AV(I–1);

Comment: j-1 indicates how many non-constants precede the input non-constant y in the $(I-1)^{st}$ section of DB;

(4) Let rel(j) be the relative position in the buffer of the j^{th} 1-bit.

(5) LP:=rel(j) +(I–2)D;

The last step is justified by noting that there are (I–2)D elements in preceding sections of the database and the input non-constant y is in position rel(j) of the $(I-1)^{st}$ section of DB.

Consider the example we used for the forward mapping again. Let PP=4 and therefore the input non-constant y is 13. Then I=4 and we read into the buffer the $(I-1)^{st}$ block of BV. In this case j=4–3=1 which means that no non-constants appear in the $(I-1)^{st}$ section before the input non-constant y. We find rel(j)=2 which means that y is the second element in its section. The logical position LP is therefore 12.

As in the case of forward mapping, we have the following theorem.

THEOREM 2. If AV is contained in fast storage, backward mapping can be performed using one disk access.

3.7. Compression Efficiency

DEFINITION 1. The compression ratio R(X) of a given compression method X will be defined as the ratio of the total size of the database before and after it is compressed using method X.

Let B_{AV} be the number of bits of each element in AV and B_{PV} be the number of bits of each element in PV. Using the BAP compression technique, BV requires $d=\left\lceil \frac{N}{D} \right\rceil$ blocks which is S·d bits where S is the block size in term of bits, AV requires $d \times B_{AV}$ bits and PV requires an expected space of $N(1-P(c)) \times B_{PV}$ bits. Thus, the number of bits to store the compressed database by BAP is

$$NOB(BAP)=S \cdot d + N(1-P(c)) \cdot B_{PV} + d \cdot B_{AV}.$$

It is obvious that P(c) is inversely proportional to NOB(BAP). We have the following theorem.

THEOREM 3. The compression ratio of BAP is

$$R(BAP) = B_{PV} \frac{N}{NOB(BAP)}$$

We assume that B_{PV}, the size of each uncompressed element, is the same in the logical database and PV.

3.8. Experiments

We experimented with a simulated data base of 400,000 elements to find the standard deviation of the compression ratio. The results are summarized in Table 1. For a given P(c), we took groups of 12288 bits and compressed them using Golomb's method and then recorded the number of bits in the compressed vector (E.S.). From this we computed the actual compression ratio and standard deviation among the groups. As we see from the table, the standard deviation is very small and increases with P(c). In our experiments, we took a slightly smaller compression ratio (.98*R) to guarantee with a very high probability that no overflow occured. Assuming a blocksize of 4096 bits , the D computed using this compression ratio is also listed in Table 1 (Theoretical D).

We conducted some simulations to examine the size of the database that can be compressed using BAP subject to a given block size, internal memory for storing AV and required response time for searching. We looked at 3 different blocksizes (512,1024 and 4096 bytes). For each of these block sizes we looked at the compression efficiency when searching must be performed in 1, 3 or 5 block accesses of the database. We conducted these experiments for P(c)=.9 in Table 2 and P(c)=.95 in Table 3. For example, we can see from Table 2 that if the blocksize is 512 and internal memory is sufficient to hold 1000 elements of AV, we can compress a database of 8,429,570 elements and perform searching in one block access.

We also conducted experiments to examine the relationship between the compression ratio, R(BAP), and the probability P(c) of the occurrence of a constant. For each P(c) in the range 0.2 up to 0.9 (step 0.1) and P(c)=0.95, we generated a file of 400,000 elements. In this experiment the probability of a constant in each location of the database was generated independently. As expected, the results in Figure 1 indicate that the compression ratio R(BAP) increases rapidly with P(c), for P(c) equal to .95 the compression ratio exceeded 16.

4. Comparison with Other Compression Techniques

In this section, we compare the efficiency of the compression techniques BAP, *header*, *bit map* and *run-length* both from the point of view of efficiency of searching and compression efficiency.

4.1. Comparisons of the Time Complexity

We consider the time complexity of the forward mapping and the backward mapping of the four compression techniques mentioned above. We discuss the time complexities in term of accesses of secondary memory.

As we saw in Section 3, the forward mapping algorithm of the BAP compression technique requires O(1) accesses. As opposed to this algorithm, the run-length and bit-map compression techniques require O(N) accesses for forward mapping where N is the number of elements in the database. The header compression technique is much better than the run-length and bit-map compression techniques. It requires O(log (HS)) accesses where HS is the expected size of the header. Thus, with respect to the forward mapping, BAP is the best one of the four techniques, the header compression is superior to the run-length and bit map compression techniques.

As we saw in Section 3, the backward mapping algorithm of BAP's is a O(1) algorithm. In case of backward mapping, BAP compression technique is still the best one of the four techniques. The header compression technique, which requires again O(log(HS)) accesses for the back mapping, is superior to the run-length and the bit map compression techniques.

4.2. Compression Efficiency

First, we will derive the analytic expressions which represent the compression efficiency for the header, bit map, and run-length compression techniques. Then the compression efficiency of these techniques are compared to BAP.

The derivation of the analytic expressions of compression efficiency for the header, bit map, run-length compression techniques are similar to that of BAP in section 3.

4.2.1. Header Compression

In order to derive the compression ratio of the header compression technique, we need to derive the average header size of this technique.

LEMMA 1. Let DB be a logical database with size N and constant probability P(c). The average number of elements in the header of the header compression technique is

$$HS = 2(N-1)\cdot P(c)\cdot(1-P(c)).$$

Proof. We define a *break* as a consecutive pair of elements in which the first is a constant and the second is a non-constant. Each occurrence of a break represents a switch from a constant run to a run of non-constants. The probability of a pair of elements in DB to form a break is $P(c)\cdot(1-P(c))$. There are N-1 possible pairs, thus the average number of breaks is $(N-1)\cdot P(c)\cdot(1-P(c))$. And hence, the average number of constant and non-constant run pairs, is

$$(N-1)\cdot P(c)\cdot(1-P(c)).$$

Since each constant run and nonconstant run pair requires 2 elements in the header [2,3], the average number of elements in the header is

$$HS = 2(N-1)\cdot P(c)\cdot(1-P(c)).$$

Q.E.D.

Let B_{HC} be the average number of bits of each element in the header and B_{PV} be the average number of bits of each nonconstant in

the compressed database (this is also the size of each element in PV of BAP). In the header compression technique, the header size is $HS \cdot B_{HC}$ bits and the non-constants require $N(1-P(c)) \times B_{PV}$ bits. Thus, the number of bits to store the compressed database by the header compression is

$$NOB(HC) = N(1-P(c)) \cdot B_{PV} + HS \cdot B_{HC}.$$

And hence, the compression ratio of the header compression technique is

$$R(HC) = \frac{N \cdot B_{PV}}{NOB(HC)},$$

4.2.2. Bit Map Compression

In bit map compression technique, the bit map requires N bits, and the non-constants require $N(1-P(c)) \times B_{PV}$ bits. Thus, the number of bits to store the compressed database by the bit map technique is

$$NOB(BM) = N(1-P(c)) \cdot B_{PV} + N.$$

Thus, the compression ratio of the bit map compression technique is

$$R(BM) = \frac{N \cdot B_{PV}}{NOB(BM)},$$

4.2.3. Run-length Compression

For this method we assume the values in the database are drawn from a certain domain and allow the compression of any run of similar values in the database. For this reason we introduce the variable domain size (DS) of the database which counts how many different values the elements can assume. For a given value x_l in the domain, we denote by $P(x_l)$ the probability it is found in any location of the database. First, we derive the average length of runs in a given database. Let DB be a given database with size N and domain size DS.

LEMMA 2. The average length of runs in DB is

$$\overline{RL} = \sum_{l=1}^{DS} (P(x_l) \sum_{j=1}^{N} \frac{1}{1-P(x_l)}),$$

Proof : The proof follows by observing that the length of a run of x_l's is a random variable with geometric distribution where the run is terminated as soon as a non x_l appears. The expected value of this random variable is $\frac{1}{1-P(x_l)}$. The above result is obtained by summing over all elements of the domain. Q.E.D.

Let B_{RL} be the number of bits of the counter field in the run length encoding scheme, B_{PV} be the average number of bits of the record fields in the compressed database and F be the number of bits of the separator field that indicates run-length encoding follows. Assume that RN is the expected value of the number of runs. We assume that runs of size smaller than four are left uncompressed in the database, since compression only increases the length of the stored object.

The total number of bits to store the compressed database required by the run-length compression technique is

$$NOB(RL) = RN \times (B_{RL} + B_{PV} + F) + (N - RN \times \overline{RL}) \times B_{PV}.$$

The second term represents uncompressed runs in the database.

The compression ratio of the run-length compression technique is

$$R(RL) = \frac{N \cdot B_{PV}}{NOB(RL)}.$$

4.3. Comparisons of Compression Efficiency

In the following experiments, we use the assumption that the counters in AV of BAP, those in header compression and run-length encoding are all of the same size B. Using our previous notation, $B_{AV}=B_{HC}=B_{RL}=B$. We also use HC, BM and RL as a short-hand notation for the header compression, bit map compression and run-length compression technique respectively.

4.3.1. BAP and Header compressions

In this section we want to derive a condition under which the compression ratio of BAP is higher than that of HC .

THEOREM 4. If $HS > \frac{d(S+B)}{B}$ then $R(BAP) > R(HC)$ otherwise $R(BAP) \leq R(HC)$.

Proof. From section 3 and section 4, the difference between the average numbers of the bits of each value required by the two techniques is NOB(HC)-NOB(BAP), that is

$$HS \cdot B + N(1-P(c)) \cdot B_{PV} - (S \cdot d + N(1-P(c)) \cdot B_{PV} + d \cdot B)$$

The theorem follows by finding the condition on HS which makes the left hand side of the above equation greater than zero. Q.E.D.

To compare the compresstion ratio of BAP and HC in practice, we did experiments in cases of clustering and non-clustering constant runs.

In this experiment we tried to simulate clustering of runs rather than generating them independently. In many realistic data base environments we can expect such clusterings to occur. We first chose a P(c), which is the ratio of constants to all values in the data base. We then generated files of 400,000 elements for each chosen run-length and scattered the runs randomly in the database such that the overall number of zeros is consistent with the required ratio. The results of these experiments are shown in Figures 2 and 3 where B was set at 32. The ratios were set at P(c)=0.7 (Fig. 2) and P(c)=0.9 (Fig. 3). Let RL_{HC} be the experimentally generated total length of the constant run and non-constant run pairs. The goal of this experiment was to find the "break point", b, of RL_{HC} such that $R(BAP) > R(HC)$ when $RL_{HC} < b$ and $R(BAP) \leq R(HC)$ when $RL_{HC} \geq b$. We found that the breakpoint was approximately 85 for P(c)=.7 and 180 for P(c)=0.9. Both compression ratios increase with the increase in RL_{HC}.

We also conducted experiments on databases in which the generation of constants is independent between the different locations. In this case we found that the compression ratio of BAP always dominates that of HC. The relationship between the two methods is summarized in Figure 4 for various values of P(c).

4.3.2. BAP and Bit Map Compression

Since the BAP method is a derivative of bit map compression, it is clear that if we use Golomb's encoding in both cases we will end up having a slightly better compression efficiency using bit maps. For that reason we examined the effect of Golomb's method by trying one method with Golomb's compression and the other without it, as explained next.

In this experiment, eight files of size 400,000 elements were generated for P(c)=0.2, 0.4, 0.5, 0.6, 0.7 , 0.8 and 0.9 . Each file was generated with B=32 and D=4096. We compressed the files using BAP with Golomb's method and the bit map without it. We then examined the difference of R(BAP) and R(BM). Figure 5 shows that R(BM) is greater than R(BAP) when $P(c) < 0.5$ and R(BM) is smaller than R(BAP) when $P(c) > 0.5$. The difference of R(BAP) and R(BM) is very small when $P(c) < 0.7$. But when $P(c) > 0.7$, the difference increases rapidly, that is, BAP becomes much better than BM. The conclusion from this experiment is that the additional overhead involved in using Golomb's encoding is justified for bit map and BAP for databases in which P(c) is larger than 0.5.

4.3.3. BAP and Run-length Compressions

It is difficult to compare these two methods as they should be used under different environments. As we explained earlier, run-length encoding can be used in cases were many different values from the domain can be compressed where as BAP is mainly used to suppress a single prevalent constant. We can still compare under a given situation the expected compression ratio of the two methods using the analytical expressions for the compressed data base size provided earlier. Here, we only give the experimental results.

Again in this set of experiments we generated clustered runs of different sizes based on a predetermined ratio, P(c), of constants and non-constants in the data base. The file size in each case was 400,000 elements and for each test we fixed a run length and randomly scattered runs in the data base of this fixed length. The number of runs was of course determined by P(c) and the run length. For run length compression, for each specified run length ,we generated files with 5000 runs and 10000 runs for the case P(c)=0.7 and 500 runs and 1000 runs for P(c)=0.9.

The goal of this experiment was to examine the "break point", b, of run lengths such that $R(BAP) > R(RL)$ when $\overline{RL} < b$ and $R(BAP) \leq R(RL)$ when $\overline{RL} \geq b$. In this experiment, we assume that B=32 and D=4096.

Figure 6 illustrates the experimental results when P(c)=0.7. The break points are approximately 57 and 30 when RN=5,000 and 10,000. In Figure 7, we repeated the same experiment with P(c)=0.9. The break points in this case are approximately 720 and 360 when RN=500 and 1,000 respectively. We also conducted experiments in which zeroes are randomly scattered in the data base. We generated a file with domain size DS equal to 100. However, the most likely element in the file was zero which was generated with probability P(c); all the other 99 values were equally likely and generated with a total probability of 1-P(c). The run length encoding compression was allowed to compress any run of values of length more than three. The results of this experiment are shown in Figure 8. As we can see, the BAP method always dominates run-length encoding in this experiment.

5. Summary and Conclusions

In this paper, a new compression method called BAP for constant removal has been introduced. The analysis of BAP's compression effectiveness and searching complexity was developed. In order to compare BAP with other classical constant removal methods such as run-length encoding, header compression, and bit map, these methods were also analysed using similar assumptions on data characteristics. Extensive experiments were performed to validate the analytical results obtained. We experimented with databases in which runs are clustered as well as independently generated constants. We identified ranges under which a partial order of compression methods is derived in terms of the effectiveness of compression ratio.

One of the conclusions of this study is that there is no overall winner under all circumstances. However, BAP is the clear winner in many ranges of data characteristics with respect to compression ratio. In addition to compression effectiveness in terms of physical size, BAP also gives very fast searching for both forward and backward mapping, typically, just one disk access. BAP is also more flexible in that it allows the user's computing environment to be incorporated to achieve a more tailored solution to the compression problems. For example, the available amount of main memory storage and effective block size can have direct impact on the performance of BAP both in terms of compression ratio and searching time.

One of the major disadvantages of the classical methods such as run-length encoding is that they cannot support updates to the database without completely readjusting the runs starting at the affected position all the way to the end of the file. The proponents of these methods claim that SSDBs are primarily static, but it is still an important requirement to support limited amount of updates in order to provide services such as removing outliers, adjusting scientific observations, etc. In BAP, the support of a dynamic database is provided by allowing some small percentage of free space in each block. Since each block is an independent unit of compression, the rest of the blocks are not affected by an overflowed block.

We are planning to perform more experiments by modelling the data clustering characteristics in finer detail, in many more different file sizes, main storage availability, block sizes, etc. to obtain a more detailed performance metric of BAP with respect to other constant removal compression methods for SSDBs.

We are also working on algorithms that can directly operate on compressed data using the method BAP without first decompressing the database. In addition to searching, operators such as transposition and aggregation are being developed on databases compressed by BAP. Results will be compared to the collection of similar algorithms of transposition and aggregation on databases compressed by header compression and run-length encoding [17].

References

1. Aronson, J., "Data Compression - A Comparison of Methods", Institute for Computer Science and Technology, National Bureau of Standards, Washington, D.C., pp. 3-5.

2. Eggers, S. J., Shoshani, A., "Efficient Accesses of Compressed Data", *Proceedings of the International Conference on Very Large Database*, 6, Montreal, 1980, 205-211.

3. Eggers, S. J., Olken, F., Shoshani, A., "A Compression Technique for Large Statistical

Database", *Proc. of the International Conference on Very Large Database,* 1981, pp. 424-434.

4. Shoshani, A., Olken, F., Wong, H.K.T., "Characteristics of Scientific Databases", *Proc. of the International Conference on Very Large Databases,* 1984, pp. 147-160.

5. Shoshani, A., "Statistical Databases: Characteristics, Problems, and some Solution", *Proc. of the International Conference on Very Large Database,* 1982, pp. 208-222.

6. Alsberg, P. A., "Space and Time Savings Through Large Database Compression and Dynamic Restructuring", *Proceeding of the IEEE,* Vol. 63, no. 8, August, 1975, pp. 1114-1122.

7. Gottlieb, D., Hagerth, S., Lehot, P., Rabinowitz, H., "A Classification of Compression Methods and their Usefulness for a Large Data Processing Center", *Proceedings of the International Conference on Management of Data,* Boston, 1979, pp. 93-101.

8. Huffman, D. A. "A Method for the Construction of Minimum Redundancy Codes", *Proceedings of IRE,* Vol. 40, September, 1952, pp. 1098-1101.

9. Aronson, J. "Data Compression - A Comparison of Methods", *ACM Transactions on Database Systems,* Vol. 4, no. 4, December, 1979, pp. 531-544.

10. Hahn, B., "A New Technique for the Compression and Storage of Data", *Communication of the ACM,* Vol. 17, no. 8, August, 1974, pp. 434-436.

11. Knuth, D. E., *The art of the Computer Programming, Volume 3: Sorting and Searching,* Addison-Wesley, Reading, Mass., 1973, pp. 401.

12. Tarjan, R. E., Yao, A. C., "Storing a Sparse Database", ul. Communications of the ACM, Vol. 22, no. 11, November, 1979, pp. 606-611.

13. Bassiouni, M. A., "Data compression in Scientific and Statistical Databases", *IEEE Transactions on Software Engineering,* Vol. SE-11, no. 10, October, 1985, pp. 1047-1058.

14. Bahl, L.R. and Kobayashi, H. "Image data compression by predictive coding II: Encoding algorithm", *IBM J. Res. Develop., 18(2), 1974.*

15. Teuhola, Jukka, "A Compression Method for Clustered Bit-Vectors", *Information Processing Letters,* Vol. 7, No. 6, October 1978.

16. Held, Gilbert, "Data Compression", John Wiley & Sons, New York, 1983, pp. 49-51.

17. Wong, H.K.T and Li, Jian-zhong, "Transposition Algorithm on Very Large Compressed Databases", *Proc. of VLDB,* Kyoto, Japan, August 1986.

Appendix: List of Symbols Used in the Paper

N: number of elements of uncompressed database.

DS: domain size of a database.

S: block size.

P(c): probability of appearance of constant c.

D: parameter chosen according to user's environment.

BAP: new compression scheme in this paper.

HC: header compression scheme.

BM: bit map compression scheme.

RL: run-length compression scheme.

BV: bit vector in BAP.

AV: address vector in BAP.

d: size of memory for AV (in number of elements).

PV: physical vector in BAP.

RN: number of runs in RL.

$\overline{RL}$: average length of runs in RL.

RL_{HC} : average length of a pair of constant run and non-constant run in HC.

B_{PV} : average number of bits of each element in PV.

B_{AV} : average number of bits of elements in AV.

B_{HC} : average number of bits of elements of the header in HC.

B_{RL} : average number of bits of counter fields in RL.

B: common size used in experiments for B_{AV} , B_{HC} , B_{RL} .

F: average number of bits of separator in RL.

R(BAP): compression ratio of BAP.

R(HC): compression ratio of HC.

R(BM): compression ratio of BM.

R(RL): compression ratio of RL.

P(c)	Expected No.of bits of compressed subvector (E.S.)	Expected Actual compression ratio	Standard Deviation (S.D.)	Theoretical D D=0.98*R*4096	S.D./E.S.
0.6	7685	1.59896	25.2252	6418	0.0033
0.7	7231.5	1.69923	22.7569	6821	0.0031
0.8	4163.9	2.95108	22.2406	11846	0.0053
0.9	2355	5.21783	19.2256	20945	0.0082
0.95	1200.09	10.2392	15.2876	41101	0.012

Table 1.

Block size (in bytes)	Memory for AV (No. of elements) (d)	Bits of compressed subvector	Parameter D	N=D*d (limitation of database size)	No. of block accesses per mapping (k)
512	500	1*4096	8429.57	4214780	1
		3*4096	25288.7	12644400	3
		5*4096	42147.8	21073900	5
	1000	1*4096	8429.57	8429570	1
		3*4096	25288.7	25288700	3
		5*4096	42147.8	42147800	5
1024	500	1*8192	16859.1	8429570	1
		3*8192	50577.4	25288700	3
		5*8192	84295.7	42147800	5
	1000	1*8192	16859.1	16859100	1
		3*8192	50577.4	50577400	3
		5*8192	84295.7	84295700	5
4096	500	1*32768	67436.5	33718300	1
		3*32768	202310	101155000	3
		5*32768	337183	168591000	5
	1000	1*32768	67436.5	67436500	1
		3*32768	202310	202310000	3
		5*32768	337183	337183000	5

P(c)=0.9 R=2.1 0.98*R=2.058

Table 2.

Block size (in bytes)	Memory for AV (No. of elements) (d)	Bits of compressed subvector	Parameter D	N=D*d (limitation of database size)	No. of block accesses per mapping (k)
512	500	1*4096	13848.6	6924290	1
		3*4096	41545.7	20772900	3
		5*4096	69242.9	34621400	5
	1000	1*4096	13848.6	13848600	1
		3*4096	41545.7	41545700	3
		5*4096	69242.9	69242900	5
1024	500	1*8192	27697.2	13848600	1
		3*8192	83091.5	41545700	3
		5*8192	138486	69242900	5
	1000	1*8192	27697.2	27697200	1
		3*8192	83091.5	83091500	3
		5*8192	138486	138486000	5
4096	500	1*32768	110789	55394300	1
		3*32768	332366	166183000	3
		5*32768	553943	276972000	5
	1000	1*32768	110789	110789000	1
		3*32768	332366	332366000	3
		5*32768	553943	553943000	5

P(c)=0.95 R=3.45 0.98*R=3.381

Table 3.

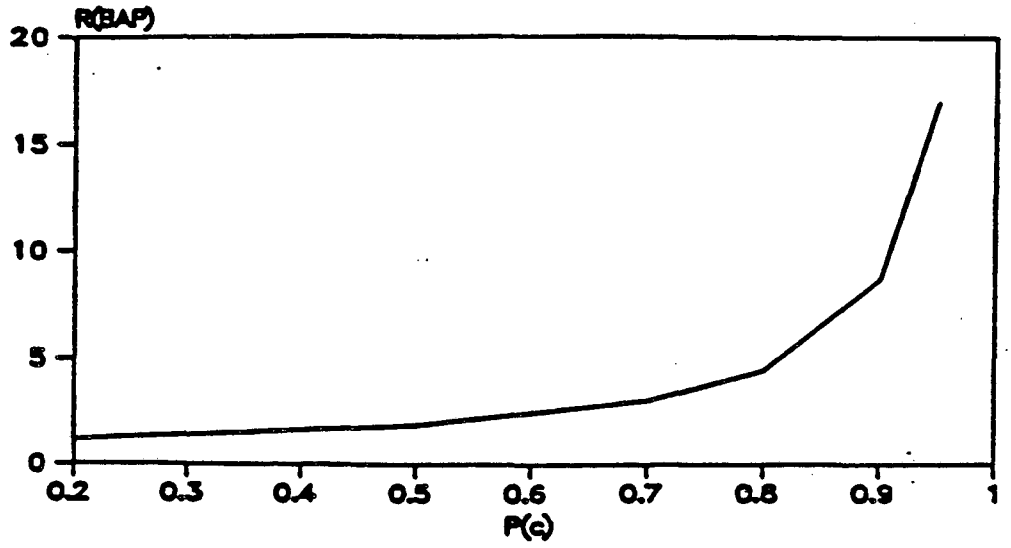

R(BAP)
20
15
10
5
0
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
P(c)

Fig. 1

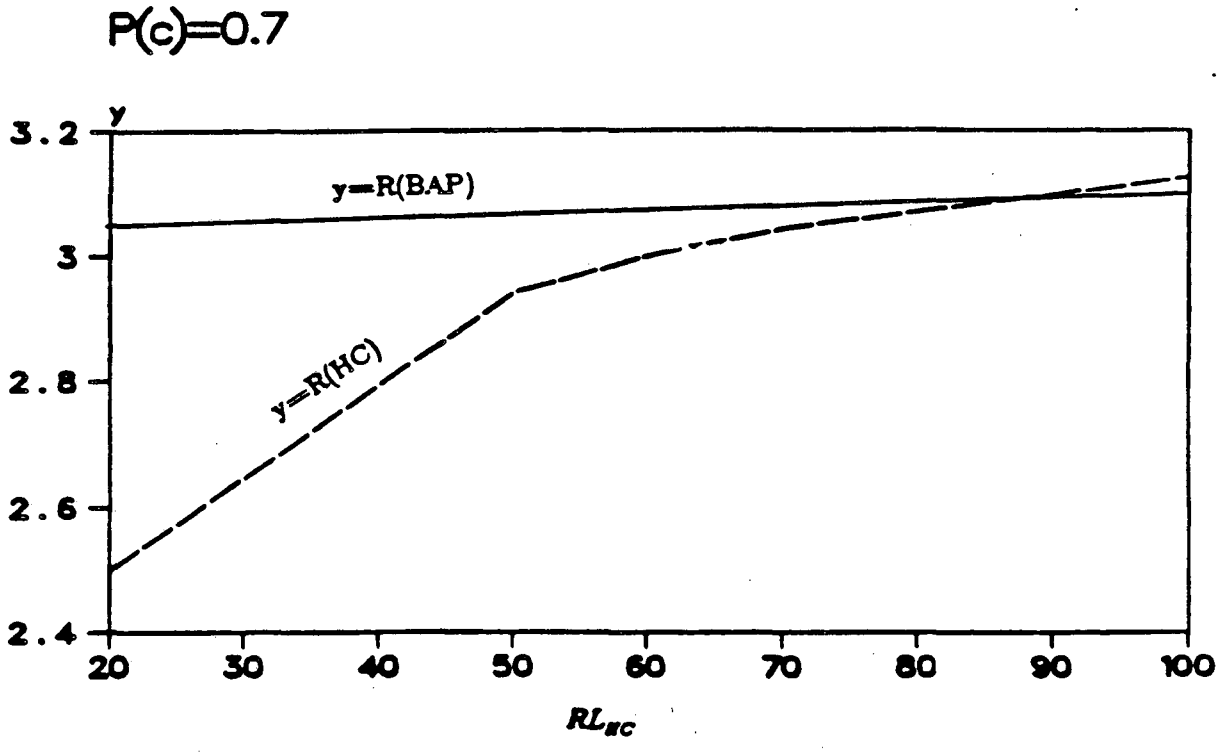

P(c)=0.7
y
3.2
3
2.8
2.6
2.4
y=R(BAP)
y=R(HC)
20
30
40
50
60
70
80
90
100
RL_{HC}

Fig. 2.

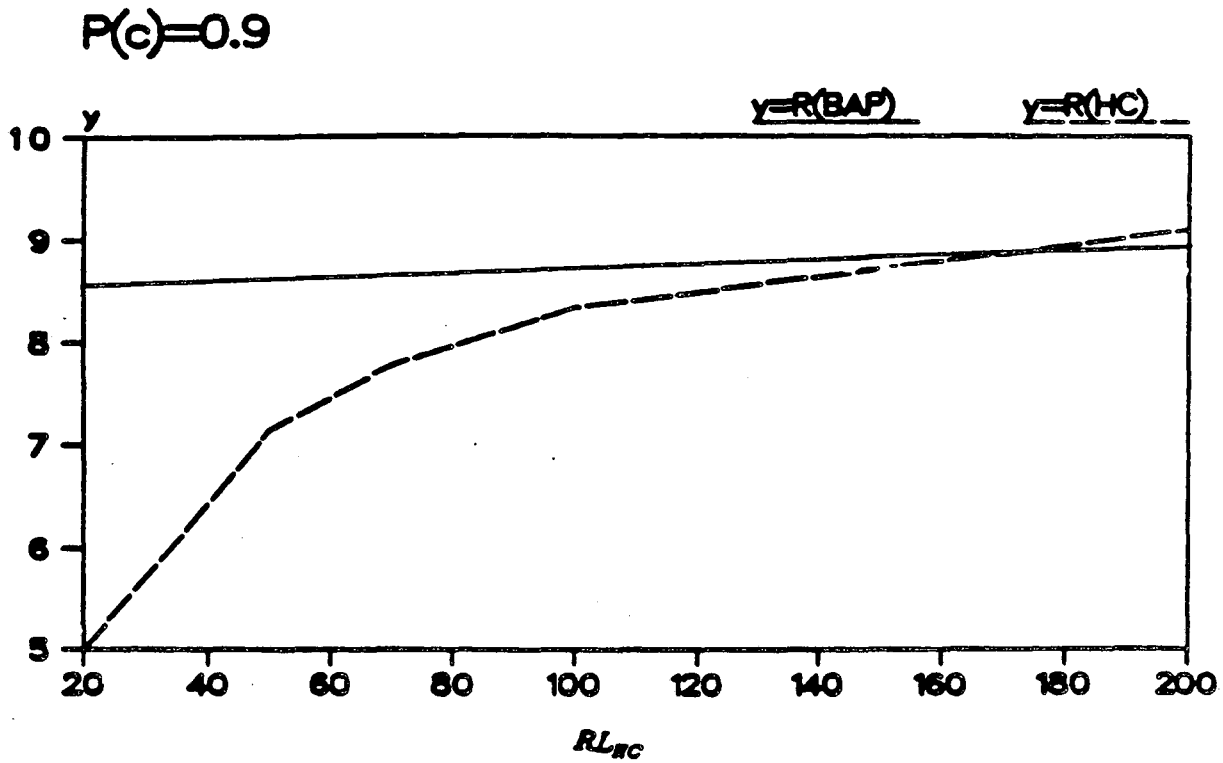

P(c)=0.9
y
y=R(BAP)
y=R(HC)
10
9
8
7
6
5
20
40
60
80
100
120
140
160
180
200
RL_{HC}

Fig. 3.

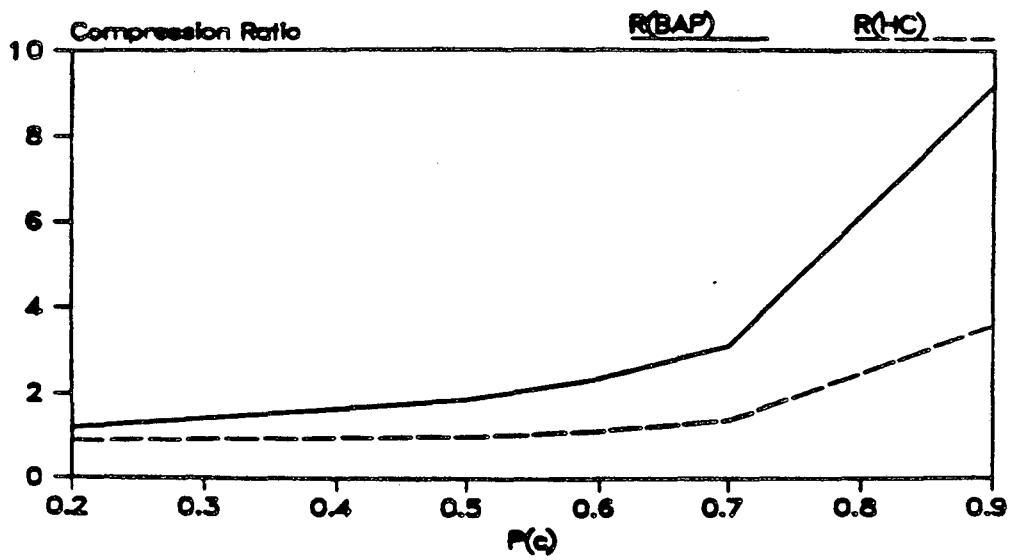

Compression Ratio
R(BAP)
R(HC)
10
8
6
4
2
0
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
P(c)

Fig. 4.

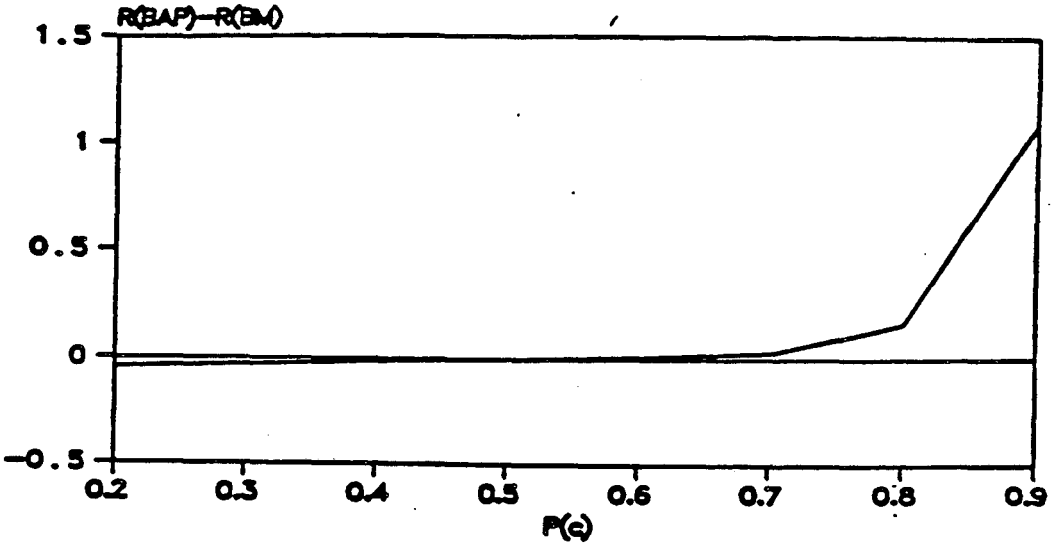

R(BAP)−R(BM)
1.5
1
0.5
0
−0.5
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
P(c)

Fig. 5.

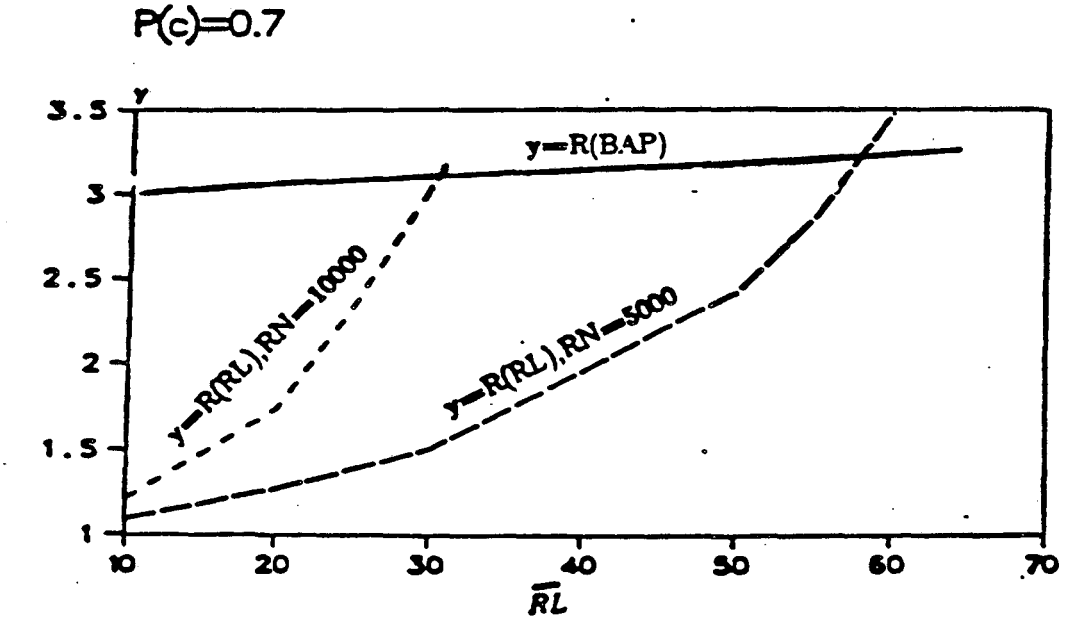

P(c)=0.7
y
3.5
3
2.5
2
1.5
1
y=R(BAP)
y=R(RL),RN=10000
y=R(RL),RN=5000
10
20
30
40
50
60
70
RL

Fig. 6.

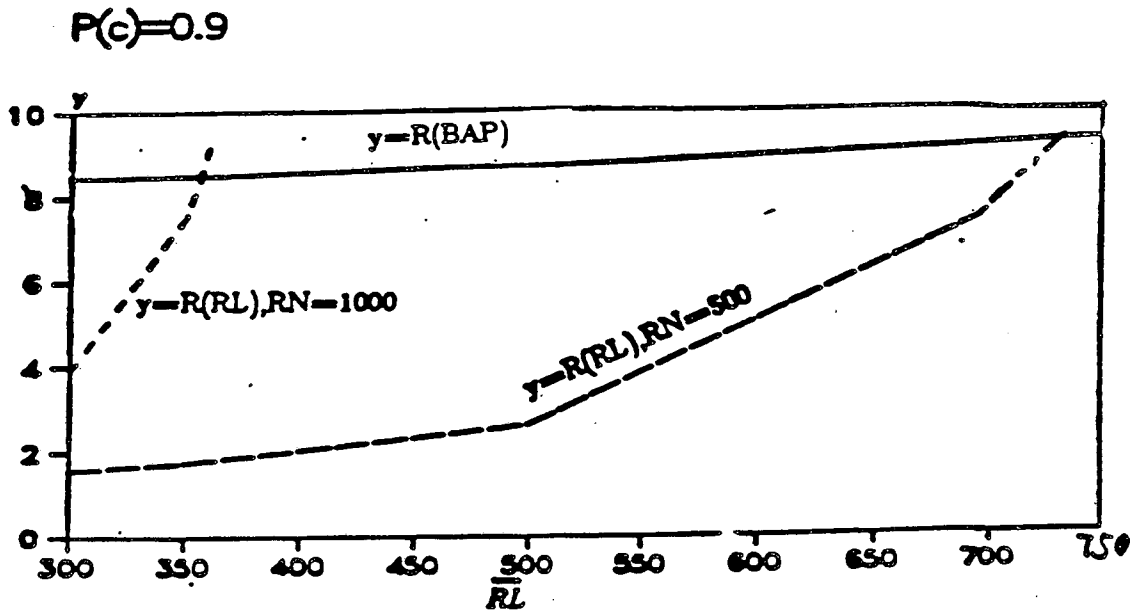

P(c)=0.9
y
10
8
6
4
2
0
y=R(BAP)
y=R(RL),RN=1000
y=R(RL),RN=500
300
350
400
450
500
550
600
650
700
750
RL

Fig. 7.

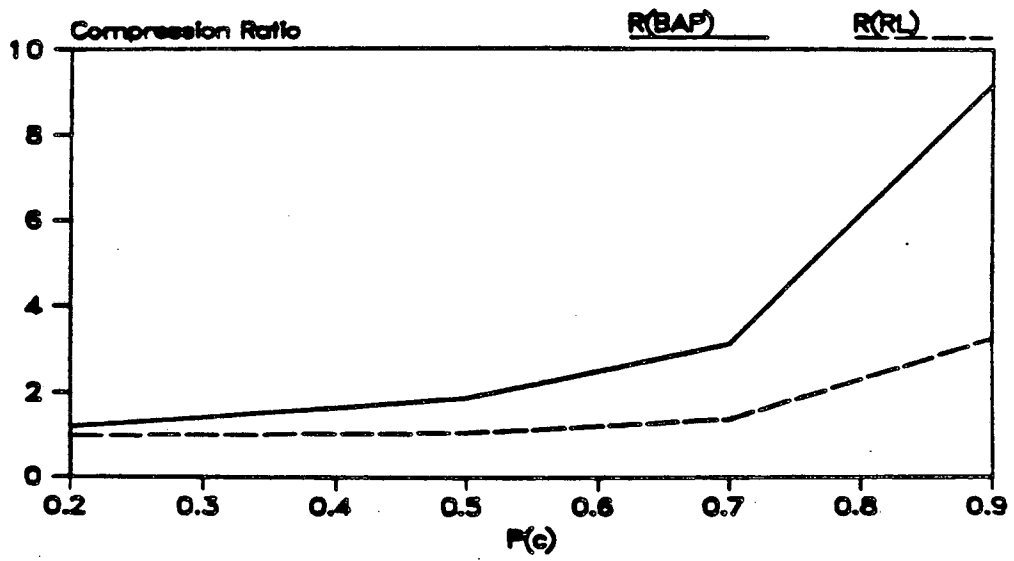

Compression Ratio
R(BAP)
R(RL)
10
8
6
4
2
0
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
P(c)

Fig. 8.

LAWRENCE BERKELEY LABORATORY
TECHNICAL INFORMATION DEPARTMENT
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720