ICEB 2005 Proceedings

International Conference on Electronic Business (ICEB)

Winter 12-5-2005

The FZ Strategy to Compress the Bitmap Index for Data Warehouses

Ye-In Chang

Chien-Show Lin

Hue-Ling Chen

The FZ Strategy to Compress the Bitmap Index for Data Warehouses[*]

Ye-In Chang, Chien-Show Lin, and Hue-Ling Chen
Dept. of Computer Science and Engineering
National Sun Yat-Sen University, Kaohsiung, Taiwan, R.O.C
E-mail: changyi@cse.nsysu.edu.tw
Tel: 886-7-5254350 Fax: 886-7-5254301

Abstract: Data warehouses contain data consolidated from several operational databases and provide the historical, and summarized data which is more appropriate for analysis than detail, individual records. Fast response time is essential for on-line decision support. A bitmap index could reach this goal in read-mostly environments. For the data with high cardinality in data warehouses, a bitmap index consists of a lot of bitmap vectors, and the size of the bitmap index could be much larger than the capacity of the disk. The WAH strategy has been presented to solve the storage overhead. However, when the bit density and clustering factor of 1's increase, the bit strings of the WAH strategy become less compressible. Therefore, in this paper, we propose the FZ strategy which compresses each bitmap vector to reduce the size of the storage space and provide efficient bitwise operations without decompressing these bitmap vectors. From our performance simulation, the FZ strategy could reduce the storage space more than the WAH strategy.

Keywords: bitmap index, compress, data warehouse, OLAP, storage.

I. Introduction

The *Data Warehouse* (DW) is a subject-oriented, integrated, time-variant, and non-volatile collection of data in support of management's decision-making process. In other words, a DW is large, special-purpose database that contains data integrated form a number of independent sources, supporting clients who wish to analyze the data for trends and anomalies. The process of analysis is usually performed with queries that aggregate, filter, and group the data in a variety of ways. OLAP is designed to provide aggregate information that can be used to analyze the contents of databases and data warehouses. Because the queries are often complex and the warehouse database is often very large, processing the queries quickly is a critical issue in the warehousing environment.

The read-mostly environment of data warehousing makes it possible to use more complex indexes to speed up queries than in situations where concurrent updates are present [19]. Bitmap indexing has been touted as a promising strategy for processing complex adhoc queries in read-mostly environments, like those of decision support systems. Most of the major commercial database systems now support some form of a bitmap index. A significant advantage of bitmap indices is that complex logical selection operations can be performed very quickly, by performing bitwise AND, OR, and NOT operations.

Data warehouses are essential for modern business to support the decision making. For various applications in data warehouses, where most of the attributes have high cardinality, the classical bitmap index produces one bitmap for each distinct value of the attribute being indexed [4] [5] [6] [8] [11] [13] [14] [15] [16] [17] [18] [19]. The size of the indices could be much larger than the size of the dataset. The main advantage for using a compressed bitmap index is to reduce the space requirement. However, the bitmaps from the bitmap indices are often very sparse; that is, they contain mostly zero bits. Moreover, most of the generic compression algorithms do not support fast bitwise logical operations, the compressed bitmap indices are usually slower in processing queries than their uncompressed counterparts. To increase the efficiency of query processing, a number of specialized compression algorithms have been developed [1][7][12][17][18][21][22]. The Byte-aligned Bitmap Code (BBC) is an example of such a strategy [1]. This strategy permits efficient operations without decompression, thereby reducing both the disk space requirement and the memory requirement for performing operations. Another specialized compression strategy called the Word-Aligned Hybrid run-length code (WAH) [21] is an efficient strategy that significantly outperforms BBC.

The WAH strategy [21] has been presented to solve the storage overhead. The main advantage of the WAH strategy is that compressed indexes are much smaller than the uncompressed ones and the average processing time is about the same [22]. However, when the bit density and clustering factor of 1's increase, the bit strings of the WAH strategy become less compressible. Therefore, we propose the Filtering-Out-Zeros (FZ) strategy to compress each bitmap vector to reduce the size of the storage space. The basic idea of the FZ strategy is to remove some continuous 0's in the bitmap vector, since there often exist large amount of 0's in the bitmap vector. For the bitmap vector with 128 bits (as shown in Figure 1-(a)), after compression, our FZ strategy stores only 40 bits (as shown in Figure 1-(b)), as compared to 96 bits in WAN strategy (as shown in Figure 1-(c)), where the length of the basic unit for compressing in FZ strategy is eight bits and that in WAH strategy is a word (= 32 bits).

Proceedings of the Fifth International Conference on Electronic Business, Hong Kong, December 5-9, 2005, pp. 125 - 131.

10000000000000000000011100000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000001111
(a)
01000000000000000000001110000000
10000000000000000000000000000011
00000000000000000000000000001111
(b)

1010000000000001 1000000000000011100001111
(c)

Figure 1: A comparison of compressed bitmap vector: (a) the bitmap vector with 128 bits; (b) the bitmap vector compressed by the WAH strategy; (c) the bitmap vector compressed by the FZ strategy.

Although it is easy to compress bitmap vectors, the query processing time is increasing on those compressed bitmap vectors. This is because the operations on the compressed bitmaps are much slower than the same operations on the uncompressed ones [22]. Therefore, we also propose the related bitwise AND and OR operations on those bit strings compressed by the FZ strategy. The time needed by one such operation on two operands is related to the size of the compressed bitmap vectors [21]. Therefore, the FZ strategy needs the smaller storage space than the WAH strategy, but also could take less time to perform the bitwise operation than the WAH strategy. In conclusion, our proposed range-based bitmap index strategies and the FZ strategy are suitable for processing attributes with thousands of distinct values.

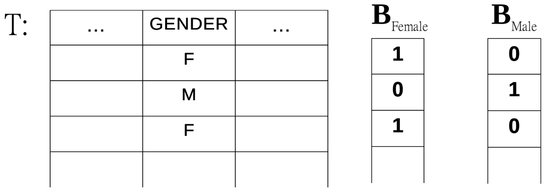
The rest of the paper is organized as follows. In Section 2, we give a survey of the strategies of bitmap indexing for data warehouses. In Section 3, we present the FZ strategy to compress the bitmap vector by filtering out many zeros to reduce the size of the storage space. In Section 4, we study the performance of the FZ strategy. In Section 5, we give the conclusion.

II. Background

Bitmap indexes were first developed for database use in the Model 204 product from Computer Corporation of America [9]. The strategy has been implemented in several commercial DBMSs (IBM, Informix, Oracle, Sybase) [3]. A significant advantage of bitmap indices over conventional hash and tree index is that complex logical selection operations can be performed very quickly, by performing bitwise AND, OR, XOR, and NOT operations supported by the hardware. And bitmap indexes can be much compact than the traditional B+ tree, especially for attributes with low cardinality [3][12]. On the other hand, tree structures, like the B-tree and R-tree, have a great drawback in the data warehouse. It is a well-known fact that tree structures degenerate when the number of dimensions is increased.

Previous strategies for constructing bitmap indexes could be classified into two classes based on the goals of the improvement: (1) time, (2) space. Basically, some of strategies are related to improve the query processing time, while some of strategies are related to improve the huge storage space. For the strategies which aim to improve the query processing time could be further classified into two groups according to query types: exact query, including *Simple Bitmap Index* [10], *Bit-Sliced Index* [10], *Encoded Bitmap Index* [23] and range query, including *Range-Based Bitmap Index* [23], *Range-Encoded Bitmap Index* and *Compressing Bitmap Index*[12][21].

The basic idea behind bitmap indexing is to use a string of bits (0 or 1) to indicate whether an attribute in a tuple is equal to a specific value or not [10][23]. The position of a bit in the string denotes the position of a tuple in the table. The bit is set, if the content of an attribute is associated with a specific value. For example, the *Simple Bitmap Index* (SBI) on an attribute GENDER, with domain {Male, Female}, results in two bitmap vectors, say B_M and B_F. For B_M, the bit is set to 1, if the corresponding tuple has the attribute GENDER = Male; otherwise, the bit is set to 0. For B_F, the bit is set to 1, if the associated tuple has the attribute GENDER = Female; otherwise, the bit is set to 0, as shown in Figure 2. The simple bitmap index on the attribute GENDER, B^{GENDER}, is the collection of bitmap vectors $\{B_M, B_F\}$.



…	GENDER	…
	F	
	M	
	F	

B_{Female}	B_{Male}
1	0
0	1
1	0

Figure 2: An example of the simple bitmap index

In the range-based index, a bitmap vector is used to represent a range, instead of a distinct value. For strategies supporting the range query, they could be further classified into two types: static and dynamic. For the static type, we mean that the range of each partition has the same width, regardless the type of data distribution. Two of well-known strategies for supporting static range bitmap indexes are the range encoding [2][20] and the interval encoding [3]. On the other hand, when the size of the range of each partition is not always the same, we call this approach *dynamic range index*. The Dynamic Bucket Expansion and Contraction strategy (DBEC) [23] dynamically construct bitmap vectors according to the types of data.

Moreover, some strategies concern to reduce the storage space needed by large number of bitmap indexes. The Byte-aligned Bitmap Code (BBC) [1] is based on the idea of run-length encoding that represents consecutive identical bits by their bit values and their lengths. Given a bit sequence, the BBC strategy first divides it into bytes, and then groups the bytes. The Word-Aligned Hybrid code (WAH) [21] is much simpler and it stores compressed data in words rather than in bytes.

WAH is similar to BBC which is a hybrid between the run-length encoding and the literal strategy. There are two types of words in WAH: *literal* words and *fill* words. WAH uses the most significant bit of a word to distinguish between a literal word (0) and a fill word (1). This choice allows one to easily distinguish a literal word from a fill word without explicitly extracting the bit. The lower bits of a literal word contain the bit values from the bitmap. The second most significant bit of a fill word is the fill bit and the lower bits store the fill length. WAH imposes the word-alignment requirement on the fills, it requires that all fill lengths be integer multiples of the number of bits in a literal word. The word-alignment ensures that logical operation functions only need to access words not bytes or bits.

Figure 3 shows a WAH bit vector representing 128 bits. In this example, each computer word contains 32 bits. Each literal word stores 31 bits from the bitmap and each fill word represents a fill with a multiple of 31 bits. If the machine has 64-bit words, each literal word would store 63 bits from the bitmap and each fill would have a multiple of 63 bits. The second line in Figure 3 shows how the bitmap is divided into 31-bit groups and the third line shows the hexadecimal representation of the groups. The last line shows the values of the WAH words. The first three words are normal words, two literal words and one fill word. The fill word 80000002 indicates a 0-fill of two-word long (containing 62 consecutive zero bits). Note that the fill word stores the fill length as two rather than 62. In other word, we represent the fill length by multiples of the literal word size. The fourth word is the active word that store the last few bits that can not be stored in a normal word, and another word (not shown) is needed to stores the number of useful bits in the active word. Each WAH word (last row) represents a multiple of 31 bits from the bit sequence, except the last word that represents the four leftover bits.

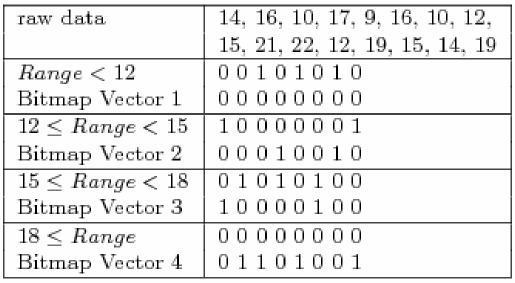
128 bits	1, 20*0, 3*1, 79*0, 25*1				
31-bit groups	1, 20*0, 3*1, 7*0	62 * 0		10 *0, 21 *1	4 *1
Group in hex	40000380	00000000	00000000	001FFFFF	0000000F
WAH (hex)	40000380	80000002		001FFFFF	0000000F

Figure 3: A WAH bitmap vector

III. The Filtering-Out-Zeros (FZ) Strategy

A bitmap index consists of a set of bitmap vectors and the size of the bitmap index could be much larger than that of the disk. This is especially true for scientific databases where most of the attributes have high cardinality. The WAH strategy [21] has been presented to solve the storage overhead. The main advantage of the WAH strategy is that compressed indexes are much smaller than the uncompressed ones and the average processing time is about the same [22]. However, when the bit density and clustering factor of 1's increase, the bit strings of the WAH strategy become less compressible. Take Table 1 as an example, we observe an interesting property that there are magnificent continuous zeros in the bitmap index. If we can filter out the consecutive 0's and only record the rest of bits, the storage space of the bitmap index could be reduced.

Table 1: An example of the range-based bitmap index

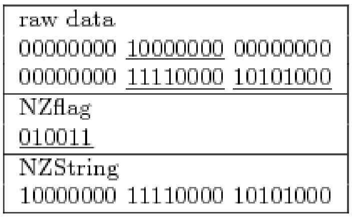


raw data	14, 16, 10, 17, 9, 16, 10, 12, 15, 21, 22, 12, 19, 15, 14, 19
$Range < 12$ Bitmap Vector 1	0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0
$12 \leq Range < 15$ Bitmap Vector 2	1 0 0 0 0 0 0 1 0 0 0 1 0 0 1 0
$15 \leq Range < 18$ Bitmap Vector 3	0 1 0 1 0 1 0 0 1 0 0 0 0 1 0 0
$18 \leq Range$ Bitmap Vector 4	0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 1

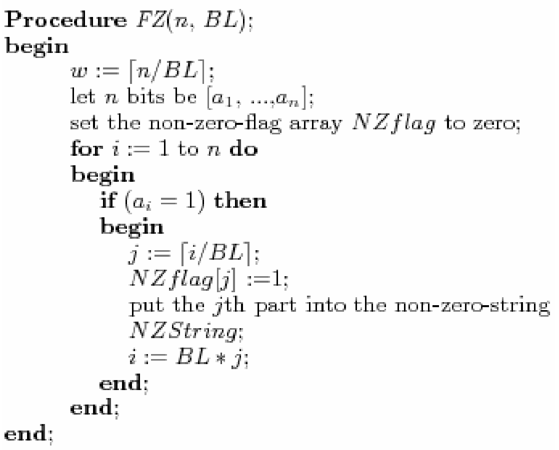
To reduce the storage space of the bitmap index, we present the Filtering-out-Zeros (FZ) strategy. We take bitmap vector 4 in Table 1 as an example to illustrate the FZ strategy. First, we divide the total 16 bits into 2 bit strings, i.e., [00000000 01101001], and the length of each bit string is 8. Next, the first bit string includes only consecutive 0's, and the second one includes 1's. Therefore, we record the first bit string by 0, and the second one by 1, i.e., the first two bits in [01 01101001]. We still record the whole bits of the second bit string, [01101001], i.e., the last eight bits in [01 01101001]. Finally, the original bitmap vector 4 [00000000 01101001] has been stored as [01 01101001] after the process of the FZ strategy. In this example, the FZ strategy reduces the storage cost from 16 bits to 10 bits.

Let's take another example shown in Table 2 to describe the FZ procedure in Figure 4. The variables used in the FZ strategy is shown in Table 3. In Table 2, the bitmap vector of raw data contains 48 bits, and every 8 bits among these 48 bits construct a bit string. Therefore, there are *w=n/BL=48/8=6* bit strings, where *n* is the number of the bits in each bitmap vector and *BL* is the length of the bit string. We use an array *NZflag* to record whether the bit string includes only consecutive 0's or not. The first, third, and 4th bit strings include only consecutive 0's in this example, and the first, third, and 4th bits of array *NZflag* are set to 0. The rest of bits of array *NZflag* are set to 1. The array *NZflag* is [010011] as shown in Table 2. According to *NZflag*, we know which bit string includes some 1's. Then, we use another array *NZString* to record the whole bits of the bit strings that include 1's. In Table 2, array *NZString* records 3 bit strings that are composed of 0's and 1's.

Table 2: An example of the FZ strategy



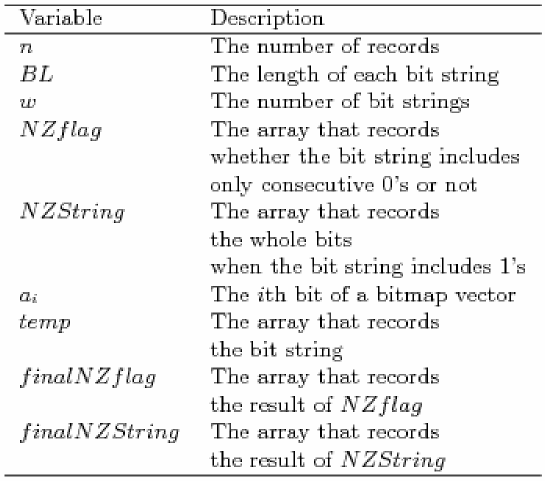
raw data
00000000 10000000 00000000 00000000 11110000 10101000
NZflag 010011
NZString 10000000 11110000 10101000



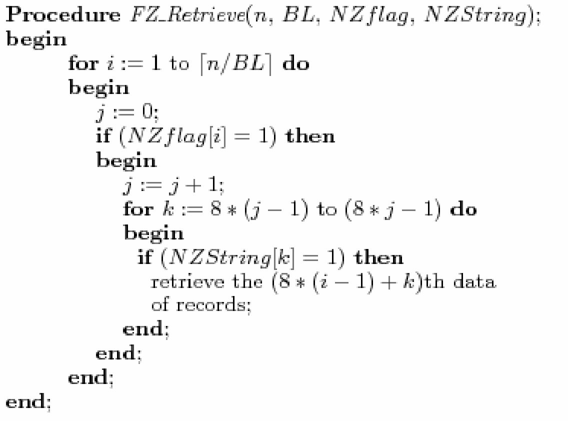
```
Procedure FZ(n, BL);
begin
    w := ⌈n/BL⌉;
    let n bits be [a_1, ...,a_n];
    set the non-zero-flag array NZflag to zero;
    for i := 1 to n do
    begin
      if (a_i = 1) then
      begin
        j := ⌈i/BL⌉;
        NZflag[j] :=1;
        put the jth part into the non-zero-string
        NZString;
        i := BL * j;
      end;
    end;
end;
```

Figure 4: The *FZ* procedure

Table 3: Variables used in the FZ, FZ_Retrieve, FZ_AND, and FZ_OR procedures



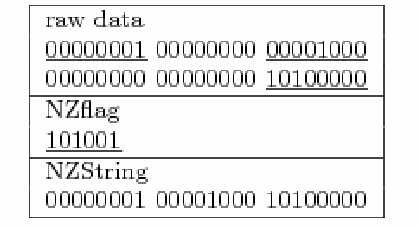
Variable	Description
n	The number of records
BL	The length of each bit string
w	The number of bit strings
$NZflag$	The array that records whether the bit string includes only consecutive 0's or not
$NZString$	The array that records the whole bits when the bit string includes 1's
a_i	The ith bit of a bitmap vector
$temp$	The array that records the bit string
$finalNZflag$	The array that records the result of $NZflag$
$finalNZString$	The array that records the result of $NZString$



```
Procedure FZ_Retrieve(n, BL, NZflag, NZString);
begin
    for i := 1 to ⌈n/BL⌉ do
    begin
      j := 0;
      if (NZflag[i] = 1) then
      begin
        j := j + 1;
        for k := 8 * (j − 1) to (8 * j − 1) do
        begin
          if (NZString[k] = 1) then
            retrieve the (8 * (i − 1) + k)th data
            of records;
        end;
      end;
    end;
end;
```

Figure 5: The *FZ_Retrieve* procedure

Table 4: An example of the *FZ_Retrieve* strategy



raw data 00000001 00000000 00001000 00000000 00000000 10100000
NZflag 101001
NZString 00000001 00001000 10100000

Figure 5 states how to retrieve data from the compressed bitmap index. For example, shown in Table 4, there are 4 1's in the bitmap vector of raw data. After the process of the FZ strategy, the resulting arrays, *NZflag* and *NZString*, are shown in Table 4. Array *NZString* records those 4 1's. We use the array *NZflag* to calculate the position of 1's. The formula is *8*(i-1)+k*, where *i* is the position of the *i*th bit string that contains 1's, and *k* is the *k*th 1's in this bit string. Therefore, the positions of the 4 bits are 8 (=0*8+8), 21 (=2*8+5), 41 (=5*8+1), and 43 (=5*8+3), respectively. According to the calculation, we directly retrieve the 8th, 21th, 41th, and 43th records, respectively.

A significant advantage of bitmap indices is that complex bitwise operations can be performed very quickly, such as bitwise AND, OR, and NOT operations. Therefore, we explain the bitwise operations, including FZ_Retrieve, FZ_AND, and FZ_OR, on bitmap vectors which are stored after the process of the FZ strategy.

Figure 6 illustrates how to perform an FZ_AND operation on those two compressed bitmap vectors. The variables used in FZ_AND operation are shown in Table 3, and the FZ_AND procedure is shown in

Figure 7. The bitmap vector in Table 2 is recorded by *NZflag1* and *NZString1*, and the bit vector in Table 4 is recorded by *NZflag2* and *NZString2*. First, we get the *finalNZflag*=[000001] by *NZflag1*=[010011] AND *NZflag2*=[101001], as shown in the left part of Figure 6-(a). Next, according to the position of 1's in *finalNZflag*, we retrieve the corresponding bit strings *temp1* and *temp2* in *NZflag1* and *NZflag2*, respectively, and add *temp3* into *finalNZString*, where *temp3* = *temp1* AND *temp2*. Since the 6th bit of *finalNZflag* in Figure 6-(a) is 1, we retrieve the bit string related to the 6th bit in *NZflag1* from *NZString1*, [10101000], to *temp1*, and that in *NZflag2* from *NZString2*, [10100000], to *temp2*, as shown in Figure 6-(b). After getting *temp1* and *temp2*, we perform an AND operation on them, i.e., *temp3* = [10101000] AND [10100000] = [10100000], as shown in Figure 6-(c), and add the bit string [10100000] into *finalNZString*, as shown in Figure 6-(d). After the process of the FZ_AND operation, we have *finalNZString* equal to [10100000], as shown in Figure 6-(d).

This strategy provides efficient bitwise operations without decompression, which reduces both the requirement of the disk space and the memory space for performing bitwise operations. In this example, we perform 4 (=6+8) bitwise AND operations on them, where *finalNZflag* includes 6 bits, and *finalNZString* includes 8 bits. If we do not use the compressed bitmap vectors, we need to retrieve

48*2=96 bits and perform 48 bitwise AND operations.

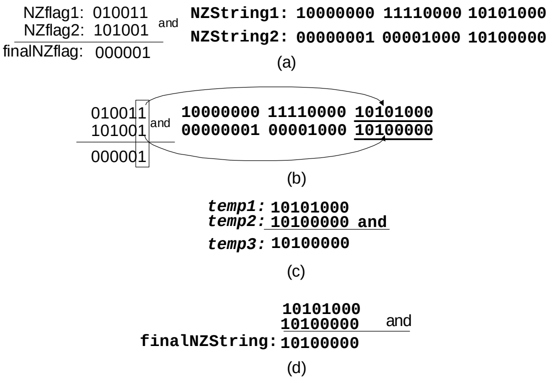
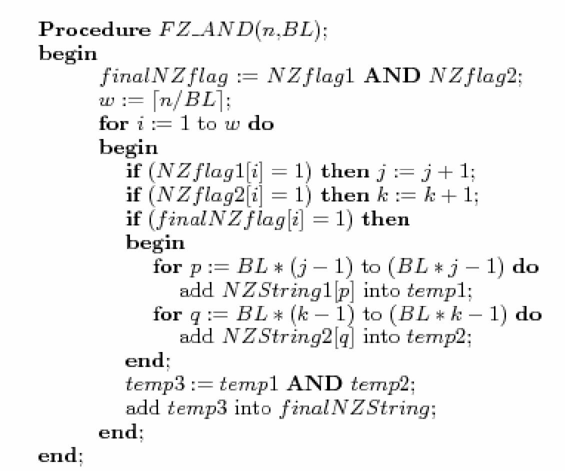


Figure 6: A bitwise FZ_AND operation on two compressed bitmap vectors by the FZ strategy: (a) the result of *finalNZflag*; (b) the bit string related to the bit in *NZString1* and *NZString2* from *NZflag1* and *NZflag2*; (c) *temp3* = *temp1* AND *temp2*; (d) the result of *finalNZString*.



```
Procedure $FZ\_AND(n,BL)$;
begin
        $finalNZflag := NZflag1$ AND $NZflag2$;
        $w := \lceil n/BL \rceil$;
        for $i := 1$ to $w$ do
        begin
            if $(NZflag1[i] = 1)$ then $j := j + 1$;
            if $(NZflag2[i] = 1)$ then $k := k + 1$;
            if $(finalNZflag[i] = 1)$ then
            begin
                for $p := BL * (j - 1)$ to $(BL * j - 1)$ do
                    add $NZString1[p]$ into $temp1$;
                for $q := BL * (k - 1)$ to $(BL * k - 1)$ do
                    add $NZString2[q]$ into $temp2$;
            end;
            $temp3 := temp1$ AND $temp2$;
            add $temp3$ into $finalNZString$;
        end;
end;
```

Figure 7: The *FZ_AND* procedure

Figure 8 illustrates how to perform a bitwise FZ_OR operation on those two bitmap vectors. We take the same two bitmap vectors, as shown in Figure 6, to perform the FZ_OR operation. The variables used in the FZ_OR operation are shown in Table 3, and the FZ_OR procedure is shown in Figure 9. First, we get *finalNZflag*=[111011] by *NZflag1*=[010011] OR *NZflag2*=[101001], as shown in the left part of Figure 8-(a). Next, according to the position of 1's in *finalNZflag*, the first, second, third, 5th, and 6th bits, we retrieve the corresponding bit strings. There are three cases according to *NZflag1* and *NZflag2*. In Case 1, the first bit of *NZflag1* is 0 and the first bit of *NZflag2* is 1. We retrieve the bit string [00000001] related to the first bit in *NZflag2* from *NZString2*, as shown in Figure 8-(b), and add the bit string into *finalNZString*, as shown in the first 8-bit string of Figure 8-(f). In Case 2, the second bit of *NZflag1* is 1 and the second bit of *NZflag2* is 0. In this case, we retrieve the bit string [10000000] related to the second bit in *NZflag1* from *NZString1*, as shown in Figure 8-(b), and add the bit string into *finalNZString*, as shown in the second 8-bit string of Figure 8-(f). In Case 3, the sixth bit of *NZflag1* is 1 and the sixth bit of *NZflag2* is also 1, we retrieve the bit string *temp1* (=[10101000]) related to the bit in *NZflag1* from *NZString1*, and the bit string *temp2* (=[10100000]) related to the bit in *NZflag2* from *NZString2*, as shown in Figure 8-(d). We finally add *temp3* into *finalNZString*, where t*emp3* = *temp1* OR *temp2*, as shown in Figure 8-(e). After eight iterations, the result *finalNZString* is shown in Figure Figure 8-(f).

In this example, we retrieve 6*8=48 bits, and perform 14 (=6+8) bitwise OR operations, where *finalNZflag* includes 6 bits, and *finalNZString* includes 8 bits in Case 3. If we do not use the compressed bitmap vectors, we need to retrieve 48*2=96 bits, and perform 48 bitwise OR operations.

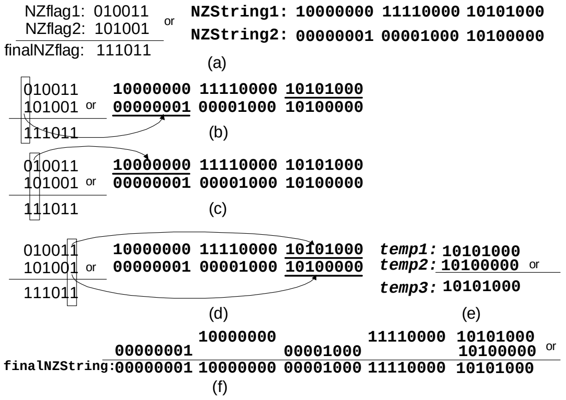
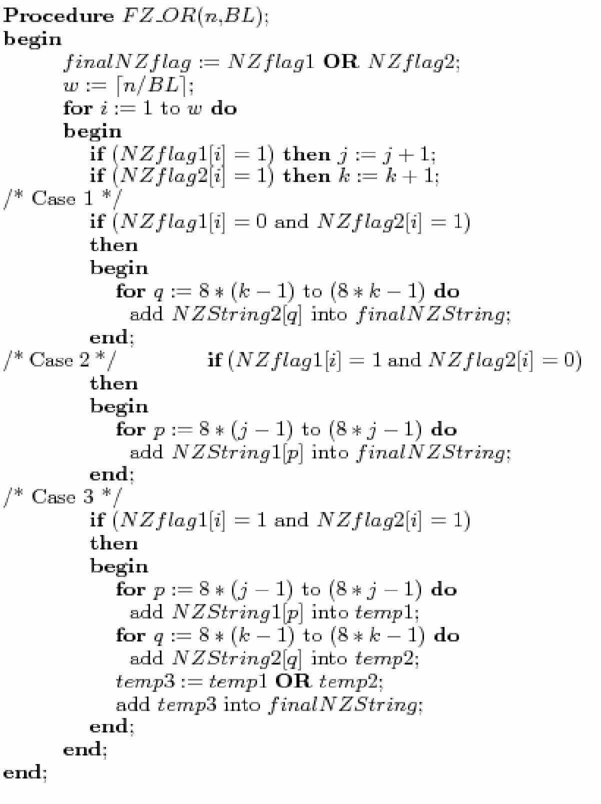


Figure 8: A bitwise FZ_OR operation on two compressed bitmap vectors by the FZ strategy: (a) the result of *finalNZflag*; (b) the bit string in Case 1 is retrieved; (c) the bit string in Case 2 is retrieved; (d) the bit strings in Case 3 are retrieved; (e) *temp3 = temp1* OR *temp2*; (f) the result of *finalNZString*.



```
Procedure $FZ\_OR(n,BL)$;
begin
        $finalNZflag := NZflag1$ OR $NZflag2$;
        $w := \lceil n/BL \rceil$;
        for $i := 1$ to $w$ do
        begin
            if $(NZflag1[i] = 1)$ then $j := j + 1$;
            if $(NZflag2[i] = 1)$ then $k := k + 1$;
/* Case 1 */
            if $(NZflag1[i] = 0$ and $NZflag2[i] = 1)$
            then
            begin
                for $q := 8 * (k - 1)$ to $(8 * k - 1)$ do
                    add $NZString2[q]$ into $finalNZString$;
            end;
/* Case 2 */            if $(NZflag1[i] = 1$ and $NZflag2[i] = 0)$
            then
            begin
                for $p := 8 * (j - 1)$ to $(8 * j - 1)$ do
                    add $NZString1[p]$ into $finalNZString$;
            end;
/* Case 3 */
            if $(NZflag1[i] = 1$ and $NZflag2[i] = 1)$
            then
            begin
                for $p := 8 * (j - 1)$ to $(8 * j - 1)$ do
                    add $NZString1[p]$ into $temp1$;
                for $q := 8 * (k - 1)$ to $(8 * k - 1)$ do
                    add $NZString2[q]$ into $temp2$;
                $temp3 := temp1$ OR $temp2$;
                add $temp3$ into $finalNZString$;
            end;
        end;
end;
```

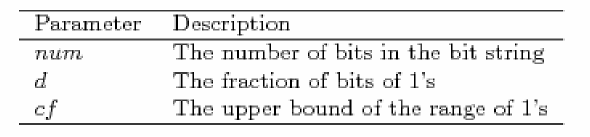
Figure 9: The *FZ_OR* procedure

IV. Performance Study

The parameters used in the performance model for the FZ strategy is shown in Table 5. Basically, we generated *num* bits, 1's or 0's, as the bitmap vector and this bit string is controlled by two parameters, the bit density and the clustering factor [21]. The performance measure in evaluating those strategies is the length, *Len*, of the bit sting after it is compressed. We plotted the average length of each bit string from experiments, and conducted 1000 experiments for each average value. The experiments were run on a Pentium 4 1.6 GHz, 256 MB of main memory, and running jdk 1.4.2 and Windows XP.

For the simulation results, the number of bit string, *num*, is bounded by 20000. In the figures to be presented as follows, the curves corresponding to the WAH and FZ strategies by changing different parameters are labeled as WAH and FZ, respectively. In Figure 10, Figure 11, and Figure 12, we compare the length of the bit string after it is compressed.

Table 5: Parameters used in the WAH and FZ strategies



Parameter	Description
num	The number of bits in the bit string
d	The fraction of bits of 1's
cf	The upper bound of the range of 1's

Figure 10 shows the length of bit strings for different *num*=10000, 12000, 14000, 16000, 18000, 20000. Each data point in this figure represents the average length of bitmap vectors with the same bit density (= 0.01) and the same clustering factor (=1). The resulting lengths constructed from both the WAH and FZ strategies increase when *num* increases. However, the length of the bit string compressed by the WAH strategy is 2 times larger than that compressed by the FZ strategy.

Figure 11 shows the length of bit strings for different values of density *d*=1/500, 1/200, 1/100, 1/50, 1/20, where *num*=10000 and *cf*=1. As the bit density increases form 1/500 to 1/20, the bit strings become less compressible and it takes more space to represent them after the compression. Moreover, the FZ strategy is more suitable for the case with the high density.

Figure 12 shows the length of bit strings for different clustering factors *cf*=0.2, 0.4, 0.6, 0.8, 1, where *num*=10000 and d = 0.01. The clustering factor, *cf*, is the fraction of the upper bound of the range of 1's. For example, when *cf* =0.2, the bits of 1's only exist from the first to 2000*th* bits in the bit string with 10000 bits. In other words, the bits of 1's are highly concentrated. As *cf* increases from 0.2 to 1, the bit string of the WAH strategy becomes less compressible and that of the FZ strategy still could be compressed well.

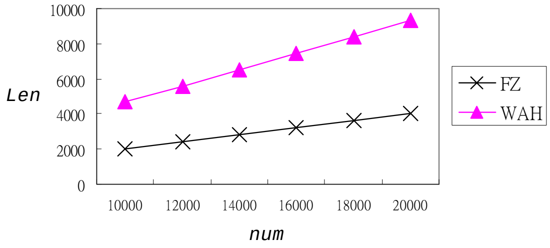


Figure 10: A comparison of the *Len* for the WAH and FZ strategies by using different numbers of bits in a string (*num*)

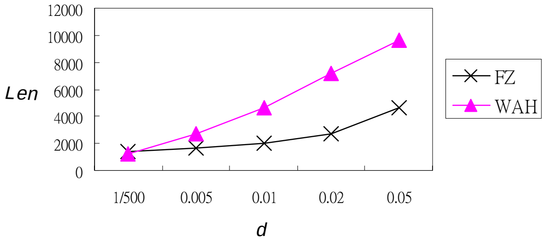


Figure 11: A comparison of the *Len* for the WAH and FZ strategies by using different values of density (*d*)

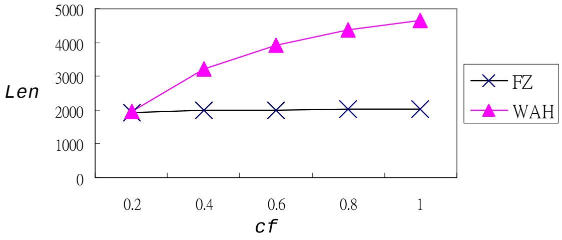


Figure 12: A comparison of the *Len* for the WAH and FZ strategies by using different clustering factors (*cf*)

V. Conclusion

To reduce the storage of the bitmap index, in this paper, we have proposed the FZ strategy which compress the bitmap vector by filtering out many zeros. We have studied the performance of our FZ strategy, and have compared it with the WAH strategy by simulation. From the simulation results, we have shown that the FZ strategy can reduce the storage cost more than the WAH strategy.

References

[1] G. Antoshenkov, “Byte-Aligned Bitmap Compression,” *Proc. of the Conf. on Data Compression*, 1995, 476.

[2] C. Y. Chan and Y. E. Ioannidis, “Bitmap Index Design and E valuation,” *Proc. of ACM SIGMOD Int. Conf. on Management*

of data, 1998, 355-366.

[3] C. Y. Chan and Y. E. Ioannidis, "An Efficient Bitmap Encoding Scheme for Selection Queries," *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, 1999, 215-226.

[4] A. Cuzzocrea, W. Wang, and U. Matrangolo, "Answering Approximate Range Aggregate Queries on OLAP Data Cubes with Probabilistic Guarantees," *Proc. of the 6th Int. Conf. on Data Warehousing and Knowledge Discovery*, 2004, 97-108.

[5] N. Koudas, "Space Efficient Bitmap Indexing," Proc. of the 9th Int. Conf. on Information and Knowledge Management, 2000, 194-201.

[6] A. Gupta , Karen C. Davis , and J. Grommon-Litton, "Performance Comparison of Property Map and Bitmap Indexing," *Proc. of the 5th ACM Int. workshop on Data Warehousing and OLAP*, 2002, 65-71.

[7] J. Li and J. Srivastava, "Efficient Aggregation Algorithm for Compressed Data Warehouses," *IEEE Trans. on Knowledge and Data Eng.*, 2002, 14(3), 515-529.

[8] Y. Lim and M. Kim, "A Bitmap Index for Multidimensional Data Cubes," *Proc. of the 15th Int. Conf. on Database and Expert Systems Applications*, 2004, 349-358.

[9] P. O'Neil, "Model 204 Architecture and Performance," *Springer-Verlag Lecture Notes in Computer Science 359, the 2nd Int. Workshop on High Performance Transactions Systems*, 1987, 40-59.

[10] P. O'Neil and D. Quass, "Improved Query Performance with Variant Indexes," *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, 1997, 38-49.

[11] T. Palpanas, "Knowledge Discovery in Data Warehouses," *SIGMOD Record*, 2000, 29(3), 88-100.

[12] A. Y. Sihem and T. Johnson, "Optimizing Queries on Compressed Bitmaps," *Proc. of the 26th Int. Conf. on Very Large Data Bases*, 2000, 329-338.

[13] N. Stefanovic, J. Han, and K. Koperski, "Object-Based Selective Materialization for Efficient Implementation of Spatial Data Cubes," *IEEE Trans. on Knowledge and Data Eng.*, 2000, 12(6), 938-958.

[14] K. Stockinger, D. Dullmann, W. Hoschek, and E. Schikuta, "Improving the Performance of High-Energy Physics Analysis Through Bitmap Indices," *Proc. of the 11th Int. Conf. on Database and Expert Systems Applications*, 2000, 835-845.

[15] K. Stockinger, "Design and Implementation of Bitmap Indices for Scientific Data," *Proc. of the Int. Database Eng. And Applications Symposium*, 2001, 47-57.

[16] K. Stockinger, "Bitmap Indices for Speeding Up High-Dimensional Data Analysis," *Proc. of the 13th Int. Conf. on Database and Expert Systems Applications*, 2002, 881-890.

[17] K. Stockinger, Kesheng Wu, and Arie Shoshani, "Strategies for Processing Ad hoc Queries on Large Data Warehouses," *Proc. of the 5th ACM Int. Workshop on Data Warehousing and OLAP*, 2002, 72-79.

[18] K. Stockinger, Kesheng Wu, and Arie Shoshani, "Evaluation Strategies for Bitmap Indices with Binning," *Proc. of the 15th Int. Conf. on Database and Expert Systems Applications*, 2004, 120-129.

[19] P. Westerman, Data Warehousing, *Morgan Kaufmann*, 2001.

[20] H. K. T. Wong, H-F. Liz, F. Olken, D. Rotem, and L. Wong, "Bit Tranposed Files," *Proc. of Int. Conf. on Very Large Data Bases*, 1985, 448-457.

[21] K. Wu, E. J. Otoo, and A. Shoshani, "Compressing Bitmap Indexes for Faster Search Operations," *Proc. of the 14th Int. Conf. on Scientific and Statistical Database Management*, 2002, 625-658.

[22] K. Wu, E. J. Otoo, and A. Shoshani, "Compressed Bitmap Indices for Efficient Query Processing," *Technical Report LBNL-47807, Lawrence Berkeley National Laboratory, Berkeley, CA*, 2001.

[23] M. C. Wu and A. P. Buchmann, "Encoded Bitmap Indexing for Data Warehouses," *Proc. of the 14th Int. Conf. on Data Eng.*, 1998, 220-230.

[24] K.-L. Wu and P. S. Yu, "Range-Based Bitmap Indexing for High-Cardinality Attributes with Skew," *Proc. of the 22th Int. Conf. on Computer Software and Applications*, 1998, 61-67.

---

[*] This research was supported in part by the National Science Council of Republic of China under Grant No. NSC-94-2213-E-110-003.