Binary Encoded Attribute-Pairing Technique for Database Compression

Akanksha Baid and Swetha Krishnan
Computer Sciences Department
University of Wisconsin, Madison
baid,swetha@cs.wisc.edu

Abstract

Data Compression makes for more efficient use of disk space, especially if space is expensive or is limited. It also allows better utilization of bandwidth for transfer of data. For relational databases, however, using standard compression techniques like Gzip or Zip does not take advantage of the relational properties , since these techniques do not look at the nature of the data. A large number of databases exist that do not require frequent updates and need to focus more on storage costs and transfer costs. OLAP is an excellent example of the same. In this paper we focus on compression of relational databases, for optimizing storage space. We use bitwise encoding based on distinct attribute values to represent the data in the columns(attributes) of our database. We then further encode the resulting columns pairwise at each stage of our compression. We discuss various observations we have made to pair columns exploiting the properties that exist for the relation, and provide experimental results to support the same. However, one could also choose to compress the database without any prior knowledge of the data. In this case, the pairing of columns for encoding would be in the order in which they occur in the database. For evaluation purposes, we compare our scheme to both Gzip and Zip at every stage, and find that our scheme reduces the database size by a greater extent in most cases.

1 Introduction

Compression is a heavily used technique in many of today's computer systems. To name just a few applications, compression is used for audio, image and video data in multi-media systems, to carry out backups. Data Compression is the process of encoding a given data set by applying some algorithm to it that produces output that requires fewer bytes than required by the original data. The compressed version of the data would require less storage space than the original version. This enables more efficient use of disk space, especially if space is expensive and or is limited.

Database sizes have increased faster than the available bandwidth to transfer the data. Good compression techniques allow transferring more data for a given bandwidth. Compression schemes have not gained widespread acceptance in the relational database arena because most database operations are CPU intensive. The CPU costs of compressing and decompressing a database would be too frequent and too exorbitant in this case. For the most part, people still tend to use general purpose compression tools like zip and gzip for compression of databases as well. These tools however, do not exploit the properties of the 'relational' database. A large number of databases exist that do not require frequent updates and for which storage costs and transfer costs are more of an issue than CPU costs. Data warehousing is an excellent example of the same. In this paper we target databases that are not updated very often. We focus on optimizing storage space.

We use bitwise encoding based on distinct attribute values to represent the data in the columns of our database. We then further encode the resulting columns pairwise at each stage of our compression. The number of stages of compression to be applied to a database depends on the nature of the data distribution and the correlations amongst the columns, and is configurable. We discuss various observations we have made to couple columns intelligently by examining the properties that hold for the relation, and provide experimental results to support the same. However, one could also choose to compress the database without any prior knowledge of the data. In this case the columns would get compressed in pairs in the order in which they occur in the database. For evaluation purposes, we compare our scheme to both gzip and zip at every stage, and find our scheme to perform better in most cases.

The rest of the paper is structured as follows. Section 2 discusses the related work and section 3 discusses our approach to database compression. We present our evaluation and results in section 4, including some observations and possible optimizations to our scheme in the same section. We present our conclusions in section 5.

2 Related Work

There are a number of standard approaches to data compression, such as those used by the Linux utilities Gzip [3] and Zip. Gzip compresses only one file and does not have a header while Zip contains header information about what files are contained in the file. The deflation algorithm used by Gzip (and also Zip) is a variation of the Lempel-Ziv algorithm. It finds duplicated strings in the input data. The second occurrence of a string is replaced by a pointer to the previous string, in the form of a pair (distance,length). Distances are limited to 32K bytes, and lengths are limited to 258 bytes. When a string does not occur anywhere in the previous 32K bytes, it is emitted as a sequence of literal bytes. Literals or match lengths are compressed with one Huffman tree, and match distances are compressed with another tree. The trees are stored in a compact form at the start of each block. Duplicated strings are found using a hash table.

The approach used by Gzip and Zip is not tailored for database compression. The algorithm fails to exploit any features of a relational database that allow it to be treated differently from a raw data file and thus enable better compression. Our approach is specifically targeted at relational databases, where relationships between the columns can be exploited for compression.

A more recent approach to compression is that taken by C-Store [1], a column-oriented database. C-store compresses columns using one of four encodings chosen depending upon the 'ordering' (sorted/unsorted) of the values within a column, and depending upon the proportion of distinct values contained in the column. The four encodings it uses are run-length encoding (for self-ordered, few distinct values columns), bitmap encoding (for foreign-ordered, few distinct values columns), delta encoding (for self-order, many distinct values columns) and standard utilities like Gzip (for foreign-order, many distinct values columns).

Our approach too is similar to C-Store in that it looks at distinct values within columns. However, our approach consistently uses binary encoding of columns as opposed to using different encoding schemes. It is yet to be seen how our approach compares with C-Store in the extent of compression achieved.

3 Our Approach to Compression

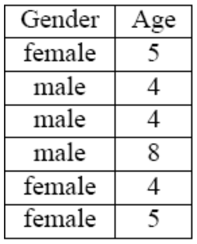
We integrate two key methods, namely binary encoding of distinct values and pairwise encoding of attributes, to build our compression technique. These are discussed in the following subsections.

3.1 Binary Encoding of Distinct Values

We base our compression technique on the observation that a relation in an RDMS generally contains a limited number of distinct values for each attribute and these values repeat over the huge number of tuples present in the database. Therefore, binary encoding of the distinct values of each attribute, followed by representation of the tuple values in each column of the relation with the corresponding encoded values would transform the entire relation into bits and thus compress it. This approach is useful for storage purposes and is not intended for databases that are written to very frequently, since the compelete binary encoding would have to be revised if the write results in a new distinct value for an attribute. In essence, we focus on an approach that optimizes storage alone.

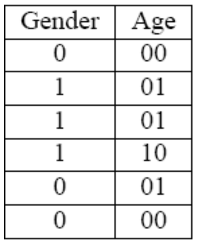
We determine the number of distinct values in each column and encode the data into bits accordingly. For example if the ’Age’ field in the ’Students’ database contains 9 distinct values we need the upper bound of log (9) i.e. 4 bits to represent each data value in the Age field. To examine the compression benefits achieved by this method, assume that ’Age’ is of type integer, with 9 distinct values and that the ’Students’ database with ’Age’ as one of the attributes contains 100 records. The storage space required to store the Age column without any compression would be 100*sizeof(int) = 400 bytes. With our compression scheme we would need 100*4(bits) = 400 bits = 50 bytes. It is easy to see that this scheme is particularly beneficial when storing data containing large strings.

The above example illustrates the compression benefit with just one column transformed into bits. We apply the same compression technique to every field in the database. Thus, the net compression achieved would be more significant. This constitutes *Stage 1* of our compression technique. Figure 2 shows the transformed columns of the Students table after *Stage 1*.



Gender	Age
female	5
male	4
male	4
male	8
female	4
female	5

Figure 1: **’Gender’ and ’Age’ columns of the ’Students’ database for illustration of our compression technique**



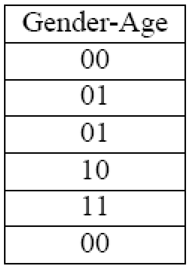
Gender	Age
0	00
1	01
1	01
1	10
0	01
0	00

Figure 2: **’Students’ table after *Stage 1* of compression.**

3.2 Pairwise Encoding of Attributes

The idea of redundancy (repetition) of values within a column in a relational table can also be extended to two or more columns. That is, it is likely that they are few distinct values of even (column1, column2) taken together, in addition to just column1’s distinct values or column2’s distinct values. We exploit this observation to find out and encode the number of distinct values among columns taken two at a time. We then represent the two columns together as a single column with pair values transformed according to the encoding.

This constitutes *Stage 2* of our compression in which we use the bit-encoded database from *Stage 1* as input and further compress it by coupling columns in pairs of two , applying the distinct-pairs technique outlined. To examine the further compression advantage acheived, let us say we couple the ’Age’ and ’Gender’ columns. As seen in Figure 1, there are 4 distinct pairs namely (female, 5), (male, 4), (male, 8) and (female, 4). We therefore need upper bound of log(4) i.e. 2 bits to represent this data. Figure 3 shows the result of *Stage 2* compression. As seen from these tables, in this example, we have reduced the storage space from



Gender-Age
00
01
01
10
11
00

Figure 3: **’Students’ table after *Stage 2* of compression.**

6(6)+ 4(6) = 60 bytes to 6*2 = 12 bits after *Stage 2* [1]. All columns are coupled in pairs of two in a similar manner. If the database contains an even number of columns this is straightforward. If the number of columns is odd, we can either choose to leave the last column uncoupled in all cases or couple columns ’intelligently’ so as to allow maximum compression. The latter case is discussed further in the section 4

Stage 2 can also be extended to further stages, that is, pairwise-encoded columns of *Stage 2* can be further paired and compressed in *Stage 3* and so on. This is also illustrated in section 4.

4 Evaluation

We used our techniques on two databases - the ’Adults’ database containing 30162 records and the ’Names’ database containing 4540 records. The ’Adults’ database consisted of numeric data only. The size of the database without any compression was 357110 bytes (without any delimiters). Our current implementation writes data as bytes instead of bits i.e. we do not actually perform dense-packing of bits within a byte at the storage level, to keep our implementation simple. The size of the resulting database after *Stage 1* was found to be 317143 bytes (without delimiters). If we were to dense-pack bits into a byte and dense-pack bytes into a storage block on disk, the resulting size of the database after *Stage 1* and without delimiters would be 105567 bytes which is around 29.56 % of the size of the initial adults database. After *Stage 2*, our resulting file was 101797 bytes in size i.e. 28.50% of the original size. [2]

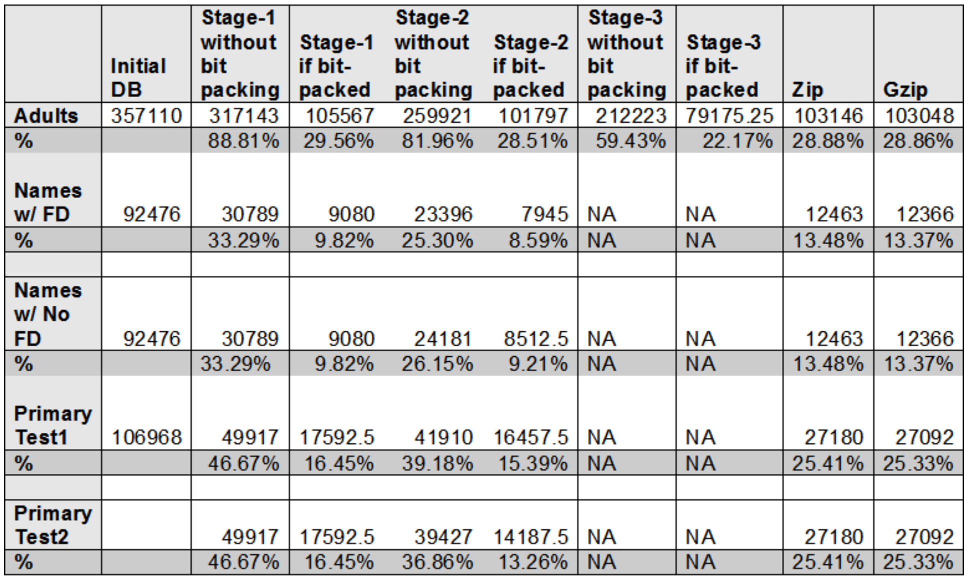
We then applied *Stage 3* compression i.e. applied pairwise encoding yet again to the paired columns obtained by processing the file in *Stage 2*. The resulting file without bit packing was 212223 bytes i.e. 59.42% of the original database file. If we were to do bit packing the resulting file would be 79175.25 bytes in size i.e. only 22.17% of the original database file.

We compared our compression results to Gzip and Zip. The reduced file size obtained on the ’Adults’ database with Gzip was found to be 103048 bytes in size which is 28.85% of the original file. With Zip, the resulting file was 103146 bytes which is 28.88% of the original file. Thus, after *Stage 2*, our compression was very close to that of Gzip and Zip for the ’Adults’ database and at *Stage 3* we achieved better compression than both Gzip and Zip by about 7%.

The ’Names’ database consisted of both string and numeric data and was a file of 92476 bytes. It is logical to assume that for long strings with few distinct values we would achieve better compression with our scheme than when we used only numeric data. We first calculate the compression for string data without any intelligent coupling of columns in *Stage 2*,i.e. we blindly couple the columns in pairs based on their position in the table. We achieved resulting files of sizes 9.82% of the original size if *Stage 1* and 9.21% of the original

[1]Assuming the maximum number of characters for a gender value is 6, we need 6 bytes for the ’Gender’ field (with sizeof(char)= 1 byte). We need 4 bytes for the ’Age’ field , taking sizeof(int) = 4 bytes. Since the example shown in Figure 1 contains 6 records, the number of bytes required, without any compression is 6*(6 bytes + 4 bytes) = 60 bytes

[2]The stated figures with dense-packing of bits within a byte have been calculated as follows: Total number of bits required = #bits for each record * #records in the database = sum total of #bits for each column(attribute) * #records in the database. The #bits for each column is calculated by taking the upper bound of the log(base 2) (#distinct values in the column).



	Initial DB	Stage-1 without bit packing	Stage-1 if bit-packed	Stage-2 without bit packing	Stage-2 if bit-packed	Stage-3 without bit packing	Stage-3 if bit-packed	Zip	Gzip
Adults	357110	317143	105567	259921	101797	212223	79175.25	103146	103048
%		88.81%	29.56%	81.96%	28.51%	59.43%	22.17%	28.88%	28.86%
Names w/ FD	92476	30789	9080	23396	7945	NA	NA	12463	12366
%		33.29%	9.82%	25.30%	8.59%	NA	NA	13.48%	13.37%
Names w/ No FD	92476	30789	9080	24181	8512.5	NA	NA	12463	12366
%		33.29%	9.82%	26.15%	9.21%	NA	NA	13.48%	13.37%
Primary Test1	106968	49917	17592.5	41910	16457.5	NA	NA	27180	27092
%		46.67%	16.45%	39.18%	15.39%	NA	NA	25.41%	25.33%
Primary Test2		49917	17592.5	39427	14187.5	NA	NA	27180	27092
%		46.67%	16.45%	36.86%	13.26%	NA	NA	25.41%	25.33%

Figure 4: **Evaluation Summary.** *Reduced File Sizes in bytes and in percentages*

file in *Stage 2*, considering dense-packing. Gzip and Zip resulted in files of sizes 13.47% and 13.37% of the original respectively.

Our results are summarized in Figure 4. The size of the file after applying various stages of compression is presented in bytes, as well as a percentage of the original file size. The first and third rows in the table shown give the results obtained for the ’Adults’ database and the ’Names’ database as discussed above. The remaining rows of the table show the results obtained by ’intelligently’ choosing columns to pairwise-encode in *Stage 2* based on certain properties of a relational database, in order to facilitate better compression. The sub-sections that follow detail these properties and how they have been exploited for better compression benefits.

4.1 Almost a Functional Dependency

Given a relation R, a set of attributes X in R is said to functionally determine another attribute Y, also in R, (written X Imp Y) if and only if each X value is associated with at most one Y value. This implies that given a tuple and the values of the attributes in X, one can determine the corresponding value of the Y attribute. It is our hypothesis that clubbing columns with relationships similar to functional dependencies would prove beneficial in compressing our database. We cannot consider functional dependencies as such because if they were present in a database it is most likely that the database would get split into more tables in order to conform to Normal form. We define *’almost’* functional dependencies (written as X Ĩmp Y) as those where the X and Y values of very few tuples violate the functional dependency condition stated above while the corresponding values of all other tuples in the relation under consideration conform to the same.

For example, as shown in Figure 5, we can see that Name Imp Gender in all cases in our ’Names’ database except for two cases where a name could map to either gender i.e. (both (Tejinder, male), (Tejinder, female) and (Lakshmi, male), (Lakshmi, female) exist in our data). Therefore according to our definition Name Ĩmp Gender (is *’almost’* a functional dependency). The number of distinct values in each column of the ’Names’ database is shown in Figure 6

As a test we compressed the ’Names’ database first by ’intelligently’ clubbing (Name ,Gender) and (Age , Country) in *Stage 2*. The distinct values for this test (Test 1) are shown in Figure 7. Then we conducted

Name	Country	Gender	Age
Amy	Macedonia	female	5
Tom	Madagascar	male	4
James	Malawi	male	6
Jack	Malaysia	male	8
Tejinder	Nepal	male	7
Lakshmi	Micronesia	female	12
Lakshmi	Maldives	male	15
Tejinder	Mali Malta	female	16

Figure 5: **Illustrating 'Almost' Functional Dependencies.** *Relation with an 'Almost' FD (Name Ĩmp Gender)*

Column name	#Distinct
Name	19
Gender	2
Country	19
Age	19

Figure 6: **Number of Distinct Values in each column of the relation that has a '*almost*' FD.**

Pair	#Distinct
Name,Gender	22
Age, Country	312

Figure 7: **Illustrating Compression exploiting '*almost*' FDs - Test 1 results after Stage 2.**

Pair	#Distinct
Name, Country	285
Age, Gender	35

Figure 8: **Illustrating Compression without exploiting '*almost*' FDs - Test 2 results after Stage 2.**

a second test (Test 2) by 'blindly' pairing (Name ,Country) and (Age, Gender) respectively. The number of distinct values for this pairing of columns is shown in Figure 8.

As seen from these Figures, coupling Name and Gender (Test 1), in *Stage 2* yielded fewer distinct (Name, Gender) tuples. The number of tuples in this case i.e. 22 was nearly equal to the number of distinct values i.e. 19 in the Name field ,which is in accordance with the *'almost'* a FD relationship between Name and Gender.

Pairing Name and Country (Test 2) on the other hand yielded #distinct values(285) somehwat towards a cross product(19*19=361) of the two. This can be explained by considering a dataset pattern as follows: Consider that we have 100 countries and a few names that are common across say 80 of these countries. For example, the name 'Mohammed' being the most common name in the world is likely to occur in a large number of countries. A database with such a data pattern would result #tuples nearly equal to [Country] * [Name] [3] if Name and Country were to be paired.

We started with the 'Names' database of 92476 bytes, and Test 1 reduced it to 23396 bytes while Test 2 could reduce it only to 24181 bytes. This shows that pairing columns with *'almost'* functional dependencies (like in Test 1) yields better results than 'blind' pairing.

An observation here though, is that the difference in compression achieved between the two tests is not much (resulting database size = 25.30% of the original size in Test 1 vs. 26.21% of the original size in Test

[3]The notation [col] stands for the number of distinct values in column 'col'

2 that is only about 1%). This is because the columns *not* related through *'almost'* functional dependency, i.e. (Age,Country) had a lot of redundant pairs as well. To elaborate upon this, we can see that while (Name, Country) pairing significantly increased the number of distinct tuples compared to (Name, Gender), the effect of this was balanced out by (Age, Gender) having much fewer distinct pairs compared to (Age, Country). This goes to show that the choice of whether or not to pair columns with *'almost'* FDs on them also depends on the distribution of distinct values in the other fields of the database.

4.2 Primary Key

A primary key is an attribute that uniquely identifies a row(record) in a table. The number of records in a relational database with the primary key integrity constraint equals the number of values in the column representing the primary key attribute. With respect to *Stage 2*, coupling the primary key column with a column having a large number of distinct values would be advantageous because the resulting number of distinct tuples of the combination of the two will always equal the number of primary key values in the table. The advantage here is achieved from not allowing the column with the large number of distinct values to be coupled with some other column, resulting in a nearly a cross product number of distinct pairs(which might be greater than the number of primary key values), since this would lead to a greater number of total distinct values across all column pairs. By *not* pairing the column with greater number of distinct values with some other column, the latter column, say one which has also high number of distinct values is made available to be paired with some other column with which it has a closer correlation, with more chances of redundancy of pairs and hence lesser number of distinct pairs on the whole. We tested this hypothesis on a database(created by extending the 'Names' database) of the following schema. (Id (int) , Name (string) , Gender (string), Country (string), Code (int)). Two tests were performed; in Test 1 we did not pair 'Id' (our primary key) with any other column, while in Test 2 , we paired 'Id' with 'Name' in *Stage 2*. The resulting file sizes are shown in rows 4 and 5 respectively of Figure 4.

Our original file was 106968 bytes in size. Test 1, where we did not pair the primary key with any of the other columns in the database resulted in a file of size 46.66% of that of the original file in *Stage 1* and a file of size 39.17% of the original file in *Stage 2* without any bit packing. For Test 2, the file after *Stage 1* was again 46.66% of the original in size while the file after *Stage 2* was 36.85% of the original in size without bit packing. With bit packing, Test 1 resulted in a file that was 15.38% of the original file while Test 2 where we paired the primary key 'Id' with the Name attribute resulted in a file that is 13.26% of the original file. As observed, pairing the primary key attribute with an attribute such as 'Name' having a large number of distinct values results in compression to a greater extent than not pairing it with any column. The results thus support our hypothesis.

4.3 Columns With Very Few Distinct Values

Most databases contain some columns with very few distinct values. In our database the Gender column is an example of the same. We found that is advantageous to couple the Gender column with a column C having a large number of distinct values as opposed to a column D with very few distinct values , except in the case where the column C is the primary key (since if C were the primary key, we would be better off coupling it with a column having greater number of distinct values rather than coupling with Gender, as explained in the former sub-section). For example consider the following schema: (City, Gender, Name, Classlevel) where the number of distinct values in each column are as follows : [City] = 200, [Gender] =2 , [Name]= 500, [Classlevel] = 5

Assuming worst-case pairing, i.e number of distinct pairs = cross product due to zero redundancy , (Gender, City) and (Name, ClassLevel) would result in 200*2 + 500*5 = 2900 total #distinct tuples. Whereas coupling (Gender, ClassLevel) and (City, Name) would result in 2*5 + 200*500 =100010 total #distinct tuples. Thus, the former coupling would result in significantly lower #distinct values and hence better scope for compression in *Stage 2* than the latter. On the other hand, if City is a primary key, then coupling Gender with City would result not be beneficial as discussed above.

4.4 Performance costs

One of the more obvious costs associated with compression is the CPU overhead that it introduces. Computing power is necessary to compress (and subsequently decompress) the data. In cases where disk space is cheap and CPU cycles are not, the cost can outweigh the reward. Our scheme also faces this same tradeoff and so we need to determine the stage at which further compression is no longer feasible. At each stage we need to determine the encoding for the distinct values remaining and store it, say in a hashtable. Further there are costs involved in probing our hash table for the codes and encoding the entire database according to them .Maintaining the respective files and data structures for the codes is also an overhead. Depending on the size of the database and the number of distinct values at each stage, these costs can vary greatly. Thus, if CPU cycles are critical, one must proceed to subsequent stages of compression carefully , making sure that the benefit is not nullified by stealing CPU cycles.

5 Conclusions

Our work focuses solely on the extent of compression achievable by using bitwise encoding based on distinct values in each column and further encoding pairs of column values . We show how coupling of columns when handled 'intelligently' tends to be most effective. In particular we found that in most cases it is beneficial to couple the primary key with the column having the maximum number of distinct values. Also, columns with very few distinct values should be paired with columns with a large number of distinct values unless the latter is the primary key. Relationships between columns that are *'almost* functional dependencies should be exploited to achieve better compression. Overall, a better knowledge of the data distribution leads to better compression; however our results do indicate that the compression achieved even without prior knowledge of the data distribution is also superior in comparison with standard compression utilities such as Gzip and Zip.

It goes unsaid that some computation overhead is necessary to achieve the compression that we hope to achieve. Based on the database and the application environment being targeted, the optimum stage up to which compression is feasible and worthy also needs to be determined, i.e. we need to decide the point at which the extra compression achieved is not worth the performance overhead involved.

References

[1] M. S. Daniel. C-store: A column-oriented dbms. In *Proceedings of the 31st VLDB Conference, Trondheim, Norway*, pages 553–564, 2005.

[2] G. Graefe and L. D. Shapiro. Data compression and database performance. In *Proc. ACM/IEEE-CS Symp. on Applied Computing*, Kansas City, MO, 1991.

[3] J. loup Gailly and M. Adler. Gzip. http://www.gzip.org.

[4] G. Ray, J. R. Haritsa, and S. Seshadri. Database compression: A performance enhancement tool. In *International Conference on Management of Data*, pages 0–, 1995.