On Compressing Databases Using Multi-Field Pattern Matching

Weizhen Mao and Rahul Simha

The College of William and Mary, Williamsburg VA 23185, USA

Abstract. This paper proposes a space-compression technique for database relations when considerable redundancy in data is present. Of particular interest is the basic pattern matching mechanism by which redundancy is diminished. While standard compression techniques use short codes to replace frequently occurring data, our method takes advantage of frequently occurring groups of data (patterns) in order to maximize the degree of compression. Thus, in some ways, the pattern matching scheme is a refinement over standard encoding schemes. It is different from pattern matching in strings in that we search for patterns consisting of multiple fields. The method has achieved compression in the range of twenty to fifty percent on actual database files.

1 Introduction

There are many reasons for designing effective techniques to compress database files. The most obvious one is to simply reduce storage space in secondary and tertiary memory. A perhaps more compelling reason is to reduce the amount of space taken up by database files residing in main memory. Having as much of the database in main memory as possible results in smaller processing times than those in a system which must frequently swap in data from disks. The relatively slow growth of disk access speeds together with cheaply available main memory has prompted some changes in this direction. Yet, requiring memory-residence for large databases that were originally intended for disks often needs prohibitively expensive new investment in hardware. Thus the importance of effective compression arises. Ideally, a compression scheme should also allow data to be processed while in a compressed state.

Many databases exhibit a great deal of redundant information eminently suited to compression. Traditional data compression methods use some form of Huffman encoding, replacing frequently occurring data with short code words. In practice, one might apply this to a database file (say, a relation) by simply analyzing the data as one long sequence of record fields. Our approach is similar in that we use an encoding scheme but very different in that we seek to take advantage of frequently occurring groups of fields.

For convenience of discussion, we focus on a particular type of database file, a sequence of records in which each field of each record is a pointer to a string. Our technique applies to records with other kinds of fields, especially ones with long strings in them. However, to emphasize the difference between the kind of compression pattern matching achieves, we will assume that pointers are used to eliminate redundancy in long strings. In this sense, some compression has already occurred and we wish to find out how much additional compression is possible by virtue of recognizing groups of frequently occurring data (in this case, subgroups of pointers). More precisely, an $r \times f$ array of pointers represents a database (relation) of r records, each consisting of f fields. The pointer in the i^{th} row and j^{th} column points to the string that should appear in field j of record i in the database. Specifically, a pointer gives two pieces of information: the address of the first symbol of the string and the length of the string. Let p be the number of bits used by such a pointer. Typically, $p = 64$. The two-dimensional array of pointers thus requires rfp bits for storage. Now, we are interested in the situation where the array contains many duplicated pointers, in particular, there are pointers that appear more than once in certain fields of the array. For example, in a database used by the personnel office of a university, the field "Position Title" may have values "Full Professor", "Associate Professor", "Assistant Professor", "Instructor", "Staff", and so on. Since there are much more employees than position titles, a certain title may appear many times in the field. Moreover, some combinations, such as "Full Professor", "Computer Science" and "Male", may occur several times as a pattern.

The most common data compression schemes in past literature include textual substitution or macro encoding schemes [4, 7, 8, 10, 11, 12], which factor out duplicated occurrences of data, replacing the repeated

elements with some sort of special marker identifying the data to be replaced at that point. The papers mentioned above all assume that the source data is a string, thus any occurrence of a common substring may be replaced by a pointer pointing to the only single copy of the substring. As discussed earlier, this type of compression replaces long strings with short pointers. A survey of database compression appears in [9].

We point out a potential drawback of our method. When a pattern of a group of pointers is found, each occurrence is replaced with a very small code word that indexes into a table of "markers" (also called a dictionary [9]) that then provides the actual group of pointers. The result of applying this compression is that record sizes *vary* in the compressed file and thus, the relational structure is lost and as a result, random access of records requires a scan from the beginning. Naturally, one would not use our scheme when very small random access times are desired. Our method is best applied when the dominant processing of a file is a scan, as is the case for many spreadsheet databases. This was indeed the case in the commercial application that motivated the paper [1]. Note that, an uncompressed file might partly lie on disk and thus a random access which might require disk I/O may be considerably slow. Our scheme is entirely appropriate for secondary and tertiary storage, from which data can be decompressed as and when it is read.

2 Multi-Field Compression

The compression scheme we will present is called *multi-field compression*, simply because it uses the idea of multi-field pattern matching. As a first step, we define a *marker index* to be a short code that replaces a particular frequently occurring group of pointers. This value indexes into a *marker table*, each entry of which is a *marker*. To be specific, a marker contains two components: a group of pointers preceded by a field position indicator, which has a few bits that indicate the fields (in the original database file) to which the marker applies. If there are l pointers in a markers, then the marker occupies $f + lp$ bits, with the last lp bits storing l pointers, and the first f bits indicating the field positions of these l pointers. For example, if $f = 4$ and the pointers are in the second and the third fields, then the first 4 bits of the marker are 0110. Note that the number of 1's in the indicator should be the same as the number of pointers in the marker.

Once a marker table is given, a compressed file of the pointer array can be constructed. In the compressed file, records and fields within records can no longer be accessed directly by indices. Instead, the encodings of the records (rows of f pointers) may have various lengths and are stored consecutively one after another. Given a marker, we say that a record is *compatible* with the marker (or vice versa) if the record contains the pointers in the fields indicated by the marker. The encoding of a record has the following format: the first $\log f$ bits to store the number of markers compatible with the record, followed by indices of the compatible markers, followed by the pointers that are not replaced by markers. If there are m markers in the marker table, each marker index requires $\log m$ bits. Intuitively, when $\log m$ is much less than p, which is often the case, it pays off to use this encoding scheme.

Fig. 1 shows an example of multi-field compression, where a pointer array, its compressed file, and the marker table are given. The pattern consisting of a and b in the first two fields and c in the fourth field occurs four times. Thus, a marker is created (in this case, the first marker in the marker table), and the four records which contain the pattern are compressed: the occurrence of the three pointers is replaced by a marker index. Note that the fields do not have to be consecutive fields and that several markers may apply to a single record (e.g. record 3 uses two markers). Before compression, the pointer array uses $30p$ bits. After compression, the number of bits used becomes $17p + 29$. When $p = 64$, the space saved is $(30p - 17p - 29)/30p = 42\%$. We also point out that in the compressed file, given the beginning address of a record, we can easily figure out the beginning address of the next record. This requirement is necessary not only for scanning the file but also for the decompression procedure.

The possibility of using multi-field compression raises some obvious questions: how does one find frequently occurring patterns? And, with various choices for the patterns, which ones should be used? We will present an algorithm that generates a multi-field compression. However, before doing so we first describe the pattern finding problem formally – some of the notations will be useful in presenting the algorithm.

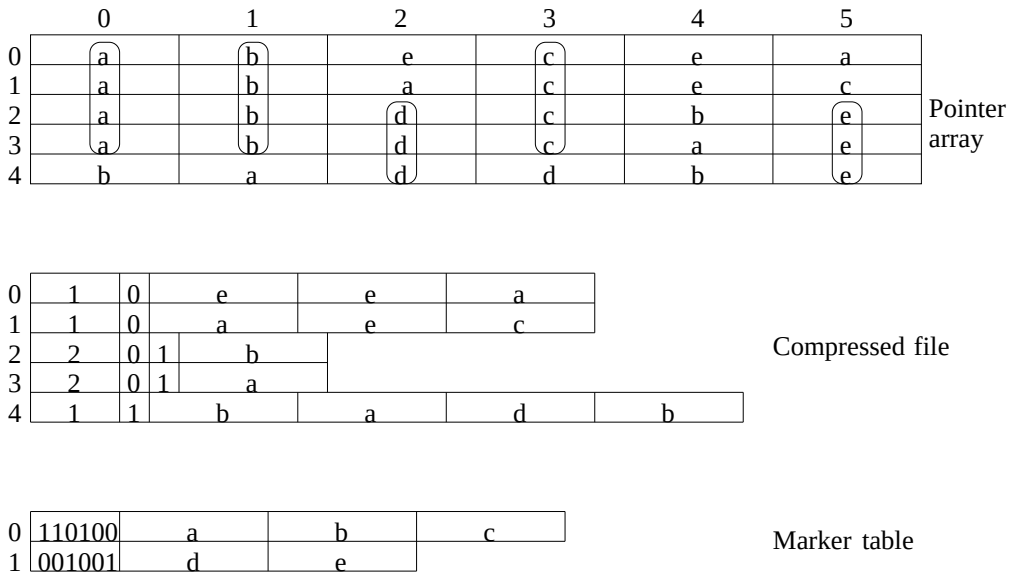

Fig. 1. Multi-filed compression

3 Formal Definition of Problem

Multi-field compression presents several choices of patterns for markers and leaves open the question of which markers are to be used for a given compatible record. These choices are to be made to maximize overall compression, a problem that we now formulate precisely.

- R is a set of *records* labeled $\{0, 1, \ldots, r-1\}$.
- F is a set of *fields* labeled $\{0, 1, \ldots, f-1\}$.
- P is a set of pointers, where each pointer uses p bits.
- $D : R \times F \rightarrow P$ is a *database*, i.e., $D(i, j) \in P$ for any $i \in R$ and $j \in F$.
- A *marker* M contains two components, $M = (M.I, M.L)$. The first component $M.I$ is the *marker indicator*, which is a function mapping from F to $\{0, 1\}$. Let $M.I(j)$ denote the j-th bit of the indicator. The *size* of the marker, $S(M)$, is defined to be the number of 1-bits in the indicator. The second component $M.L$ is the *marker list* of $S(M)$ pointers, the order of which matters.
- A marker M is *compatible* with record $i \in R$ (or vice versa) if the k^{th} pointer (for $k = 1, \ldots, S(M)$) in $M.L$ occurs in the same field position in record i as the k^{th} 1 in $M.I$. For instance, record *abbac* is compatible with marker $(01011, bac)$.
- Let $K(M)$ be the number of records compatible with marker M.
- An *assignment* is a function $A : R \rightarrow \{sets\ of\ markers\}$ for database D such that $i \in R$ is compatible with any marker in $A(i)$. Note that $|A(i)| \leq f$.
- For assignment A, define $\mathcal{M}_A = \{M : M \in A(i) \text{ for some } i \in R\}$. $\mathcal{M}_A$ is in fact the marker table.
- For assignment A and any $M \in \mathcal{M}_A$, define $C_A(M) = |\{i \in R : M \in A(i)\}|$. Note that $C_A(M) \leq K(M)$.
- For assignment A, the space in bits used by the marker table is $space_1(A) = f|\mathcal{M}_A| + (\sum_{M \in \mathcal{M}_A} S(M))p$, and the space in bits used in the compressed file is $space_2(A) = r \log f + (\sum_{M \in \mathcal{M}_A} C_A(M)) \log |\mathcal{M}_A| + lp$, where l is the number of pointers in D that are not replaced by markers under assignment A. Clearly, $l \geq rf - \sum_{M \in \mathcal{M}_A} (C_A(M)S(M))$. Finally, define $space(A) = space_1(A) + space_2(A)$.

The optimization problem that captures the theoretical interest of the data compression problem discussed in previous sections is then to determine the optimal assignment A minimizing $space(A)$ given a two-dimensional array D. How difficult is this problem? We strongly suspect that it is NP-complete, i.e., no

polynomial-time algorithm can be found at this time to solve the problem optimally [3, 5]. However, we are unable to establish such a proof. Instead, we next present a heuristic algorithm.

4 The Marker Splitting Algorithm

In this section, we describe an algorithm, called the *marker splitting algorithm*, that attempts to approximate the optimal set of markers (marker table) by iteratively adding new markers to a marker set. It is so named for the manner in which new markers are created: we start with markers containing highly correlated and repetitive fields and add markers by "splitting" them off from the existing set of markers. In this manner it is hoped that candidate patterns for compression, those that occur frequently together, will be found.

We use the notation (j, v) to denote a field-value pair, i.e., when field j has value (pointer) v. The first step in our procedure is briefly described as follows. We scan the database to identify candidate fields for inclusion in markers. Clearly, we do not wish to include fields with a large multiplicity of values, but those in which a few values repeat frequently. Accordingly, let α_{jv} denote the number of occurrences of (j, v) in the database. Next, for each combination (j_1, v_1) and (j_2, v_2) we compute the number of records in which they occur simultaneously; let $C_{j_1v_1j_2v_2}$ denote this number. Now, computing $C_{j_1v_1j_2v_2}$ for every pair of field-value combinations is clearly going to be time consuming (it requires several scans of D). However, not all field-value combinations are likely to occur frequently; thus, it is not necessary to compute $C_{j_1v_1j_2v_2}$ for infrequent combinations.

Marker Splitting Algorithm.
Input: An $r \times f$ array of pointers.
Output: An assignment A and marker table $\mathcal{M}_A$.

Find j_1, v_1, j_2, v_2 that maximizes $C_{j_1v_1j_2v_2}$;
$G \leftarrow F - \{j_1, j_2\}$;
Let M_1 be the marker of pointers v_1, v_2 in fields j_1, j_2, respectively;
Initialize the set of markers, $\mathcal{M}_A = \{M_1\}$;
$n \leftarrow 1$;
While $G \neq \emptyset$
    Find (j, v), $j \in G$ such that $\alpha_{jv} = \max_{i,u} \alpha_{iu}$;
    For each $M_k \in \mathcal{M}_A$ compute the match coefficient of M_k with (j, v):
        $\beta_k(j, v)$ = the number of records in which both M_k and (j, v) occur, i.e.,
            where each such record i is compatible with M_k and $D(i, j) = v$;
        Let $\delta_k = K(M_k) - \beta_k(j, v)$;
    Find t such that $\beta_t(j, v) - \delta_t = \max_k\{\beta_k(j, v) - \delta_k\}$;
    $n \leftarrow n + 1$;
    Construct a new marker M_n with field j and value v such that
        $\forall j : M_t.I(j) > 0, M_n.I(j) = M_t.I(j)$;
    Add M_n to $\mathcal{M}_A$;
    $\forall i \in R$: $A(i) \leftarrow \{M_n\}$ if M_n is compatible with i;
    $G \leftarrow G - \{j\}$;

Thus, a new field is iteratively selected as a candidate field for addition; a "best choice" marker is chosen to match the field and a new marker is created by adding in the selected field. This process is driven by frequency counts of the occurrence of different candidate fields. Note that the procedure above makes only one pass through the set of fields by successively removing fields from G. A simple improvement to the above to achieve better compression would be to try several passes of the above computation while adding markers to the marker table $\mathcal{M}_A$. The marker splitting algorithm was tested on several actual database files given by Allen in [1] of different sizes. The results for two data sets are shown in Table 1, in which we compared the space usage before and after the compression and showed the saving obtained.

Table 1. Experimental results

No. of records (Data set I)	Size before	Size after	Percent gain	No. of records (Data set II)	Size before	Size after	Percent gain
1000	10000	7133	28.6	1000	9000	4354	51.6
2000	20000	14512	27.4	2000	18000	9916	44.9
3000	30000	22874	23.7	3000	27000	15170	43.8
4000	40000	30893	22.7	4000	36000	20233	43.7
5000	50000	38838	22.3	5000	45000	25285	43.8

5 Summary

In this paper, we considered a database compression problem. Our approach was significantly different from that of standard encoding techniques. We sought to further improve on standard encodings by finding patterns of frequently occurring data and using codes for whole patterns. The choices available for patterns quickly lead to a combinatorial problem. We presented an iterative heuristic method that makes use of correlation among fields. Some experimentation with actual database files show the method to be promising. However, we caution that additional experimentation with various forms of the heuristic as well as with various applications is desired before the general technique can be fully evaluated.

For further research, we are interested in comparing our technique with other existing compression schemes. However, because of the requirement that the original database has to be a relational database of pointers, many textual substitution based techniques are ruled out for consideration. Two compression schemes that might be used under our special requirement are the ones discussed in [2, 6].

Acknowledgement

The authors wish to thank Mr. T. A. Allen of the World Bank for posing the problem and for some additional practical insights, and three anonymous reviewers for their constructive comments.

References

1. T. A. Allen. The IMIS database system. *The World Bank Technical Report*. Washington DC, 1994.
2. G. Ellis. Compiled hierarchical retrieval. In *Conceptual Structures: Current Research and Practice*. T. Nagle, J. Nagle, L. Gerholz, and P. Eklund (eds.), Ellis Horwood, 285-310, 1992.
3. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.
4. B. Hahn. A new technique for compression and storage of data. *Comm. ACM* **17**, 434–436, 1974.
5. R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*. R. E. Miller and J. W. Thatcher (eds.), Plenum Press, New York, 1972.
6. R. Levinson. UDS: A universal data structure. In *Conceptual Structures: Current Research and Practice*. W. M. Tepfenhart, J. P. Dick, and J. F. Sowa (eds.), Lecture Notes in AI 835, Springer-Verlag, Berlin, 230–250, 1994.
7. J. P. McCarthy. Automatic file compression. *Proceedings of International Computing Symposium*, 511–516, 1973.
8. A. Mayne and E. B. James. Information compression by factorizing common strings. *Comput. J.* **18**, 157–160, 1975.
9. M. A. Roth and S. J. Van Horn. Database compression. *SIGMOD Record* **22**, 31–39, 1993.
10. F. Rubin. Experiments in text file compression. *Comm. ACM* **19**, 617–623, 1976.
11. J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *J. ACM* **29**, 928–951, 1982.
12. R. A. Wagner. Common phrases and minimum-space text storage. *Comm. ACM* **16**, 148–152, 1973.