# Data Compression and Database Performance

Goetz Graefe
University of Colorado
Boulder, CO 80309-0430

Leonard D. Shapiro
Portland State University
Portland, OR 97207-0751

Appeared previously with fewer performance results
in Proc. ACM/IEEE-CS Symp. on Applied Computing,
Kansas City, MO, April 1991.

## Abstract

Data compression is widely used in data management to save storage space and network bandwidth. In this report, we outline the performance improvements that can be achieved by exploiting data compression in query processing. The novel idea is to leave data in compressed state as long as possible, and to only uncompress data when absolutely necessary. We will show that many query processing algorithms can manipulate compressed data just as well as decompressed data, and that processing compressed data can speed query processing by a factor much larger than the compression factor.

## 1. Introduction

Data compression is an effective means for saving storage space and network bandwidth. A large number of compression schemes have been devised based on character encoding or on detection of repetitive strings [2, 18], Many compression schemes achieve data reduction rates to 2.3−2.5 bits per character for English text [2], i.e., compression factors of about 3¼. Since compression schemes are so successful for network bandwidth, the advantageous effects of data compression on I/O performance in database systems are rather obvious, i.e., its effects on disk space, bandwidth, and throughput. However, we believe that the benefits of compression in database systems can be observed and exploited beyond I/O performance.

Database performance strongly depends on the amount of available memory, be it as I/O buffers or as work space for query processing algorithms. Therefore, it seems logical to try to use all available memory as effectively as possible — in other words, to keep and manipulate data in memory in compressed form. This requires, of course, that the query processing algorithms can operate on compressed data. In this report, we introduce techniques to allow just that, and demonstrate their effect on database performance.

In the next section, we briefly indicate related work. In Section 3, we identify the techniques that allow database processing algorithms to manipulate compressed data. Section 4 contains a preliminary performance analysis for a simple join query as commonly used in relational database systems. We offer our conclusions in Section 5.

## 2. Related Work

A number of researchers have considered text compression schemes based on letter frequency, as pioneered by Huffman [10, 11, 16, 17]. Other recent research has considered schemes based on string matching [22, 30, 32, 33] Comprehensive surveys of compression methods and schemes are given in [2, 18, 27]. Others have focussed on fast implementation of algorithms, parallel algorithms and VLSI implementations [12, 29].

Few researchers have investigated the effect of compression on database systems and their performance [20, 23, 25]. Alsberg considered disk time and space savings by using fixed-length minimum bit compression [1]. He also points out that more records fit into a page if the records are shorter, thus allowing more effective clustering of records used together. Toyama and Ura improved on Alsberg's compression scheme by relating the order of compressed and uncompressed values [28]. Eggers and Shoshani explored searching for and retrieving compressed data efficiently [8]. Goyal considered text compression methods that allow search (selection) on compressed data [13]. Cormack discusses compression of entire records (not fields) in the IMS database management system [6].

Other researchers have investigated compression and access schemes for scientific databases with very many constants (e.g., zeroes) [9, 19, 21] and considered a very special operation on compressed data (matrix transposition) [31].

For indices, many database systems use prefix- and postfix-truncation to save space and increase the fan-out of nodes, e.g. Starburst [15]. However, actual compression methods are typically not used at all in most database management systems. In particular, the performance effects of data compression on more frequently-used database operations such as relational join, duplicate elimination, and set intersection have not been considered to-date.

## 3. Database Processing with Compressed Data

In this section, we outline how compression can be exploited in database systems, and how standard query processing algorithms can be adapted to work on compressed data. We divided the discussion into three sections on I/O performance and buffering, transaction processing, and query processing.

The compression rates that can be achieved for any dataset depend, of course, on the attribute types and value distributions. For example, it is difficult to compress binary floating point numbers, but relatively easy to compress English text by a factor of 2 or 3 [2]. In the following, we do not require that all data is English text; we only require that some compression can be achieved. Since text attributes tend to be the largest fields in database files, we suspect that expecting an overall compression factor of 2 is realistic for many or even most database files. Optimal performance can only be obtained by judicious decisions which attributes to compress and which compression method to employ.

Most obviously, compression can reduce the amount of disk space required for a given data set. This has a number of ramifications on I/O performance. First, the reduced data space fits into a smaller physical disk area; therefore, the seek distances and seek times are reduced. Second, more data fit into each disk page, track, and cylinder, allowing more intelligent

clustering of related objects into physically near locations. Third, the unused disk space can be used for disk shadowing to increase reliability, availability, and I/O performance [3]. Fourth, compressed data can be transferred faster to and from disk. In other words, data compression is an effective means to increase disk bandwidth (not by increasing physical transfer rates but by increasing the information density of transferred data) and to relieve the I/O bottleneck found in many high-performance database management systems [4]. Fifth, in distributed database systems and in client-server situations, compressed data can be transferred faster across the network than uncompressed data. Uncompressed data require either more network time or a separate compression step. Finally, retaining data in compressed form in the I/O buffer allows more records to remain in the buffer, thus increasing the buffer hit rate and reducing the number of I/Os.

The last three points are actually more general. They apply to the entire storage hierarchy of tape, disk, controller caches, local and remote main memories, and CPU caches. If storage space on all levels is used more efficiently, bandwidth is saved when moving data up or down in the hierarchy, when moving data laterally (e.g., between memories or caches), and by achieving a higher hit rate at each level. Reducing the amount of bus traffic in shared-memory systems might also allow higher degrees of parallelism without bus saturation [14].

For transaction processing, there will probably be two main effects of compression. First, the buffer hit rate should increase since more records fit into the buffer space. Second, I/O to log devices should decrease since the log records can become shorter. Most systems already employ "log compression" by saving only log records of committed transactions when archiving log devices on inexpensive storage media (tapes). In addition to these methods, we propose an inexpensive technique to reduce log traffic earlier in the process, namely before a log record is first written to disk.

If the database contains compressed values, it is trivial to include compressed values in the log. In other words, compressing the database values also reduces the size of log records and therefore the I/O traffic to log devices. Thus, compressing database values improves I/O performance on both the primary database and the log. It is conceivable that the savings in logging alone justify the overhead of compressing database values to be entered into the database.

For query processing, we require that a fixed compression scheme be used for each attribute. The scheme should be fixed for each attribute to allow comparisons of database values with (compressed) predicate constants. Actually, it should be fixed for each domain, not each attribute, because this will allow comparing compressed values from different sources. This does not rule out dynamic compression schemes, even if it seems to on first sight. Instead of adjusting the encoding frequently, e.g., after each character as can be done in dynamic compression and decompression of transmission streams, the compression encoding can only be adjusted during database reorganization. Suitable statistics can be gathered while unloading the database, and a new encoding can be used starting when the database is reloaded. In fact, separating statistics gathering for dynamic compression schemes during unloading and compression of data during reloading eliminates the start-up and adjustment period during which dynamic compression

schemes do not work very effectively [2]. The parameters of the compression scheme used during reloading are made part of the meta-data or catalogs.

One form of compression that is already used is to encode small sets of large values. Basically this idea was introduced from the viewpoint of compression (rather than database design and normalization) in [1]. For example, instead of storing a long string-valued "color" attribute in each record in a large "parts" table, it is customary to encode colors using a small integer, save the encoding in a separate relation, and join the large table with the relatively small encoding table for queries that require string-valued output of the color attribute. Since such encoding tables are typically small, e.g., in the order of a few kilobytes, efficient hash-based algorithms can be used for the join that outperform both naive methods like nested loops join or the sort-based merge-join [5, 7, 26]. In the long run, we hope that such encodings can be administered automatically very much like indices in today's database management systems, and that they can be recommended by automated physical database design and tuning software where appropriate.

In query processing, compression can be exploited far beyond improved I/O performance. First, exact-match comparisons can be performed on compressed data. Consider a query to retrieve the record(s) with social security number "123-45-6789." If the constant in the selection predicate is compressed in the same way as the social security number in the data file, the comparisons can be performed without de-compressing attribute values from the data file. Note that exact-match index lookups are possible on compressed data if a consistent compression scheme is used.

Second, projection and duplicate elimination can be done without decompressing data. Since we assume fixed encodings for attributes and therefore for entire records, equal uncompressed records will have equal compressed images. Although the algorithms used for duplicate elimination and for aggregation and grouping are principally the same, the situation for aggregation is little more complex[1]. While the grouping attribute can remain compressed, the attribute on which arithmetic is performed typically must be decompressed.

Third, neither the join attributes nor other attributes need to be decompressed for most joins. Almost all joins are equality joins on keys and foreign keys, i.e., they put normalized and decomposed objects back together. Since keys and foreign keys are from the same domain, and since we require that compression schemes be fixed for each domain, a join on compressed key values will give the same results as a join on normal, decompressed key values. It might seem unusual to perform a merge join in the order of compressed values, but it nonetheless is possible and will produce correct results. The same arguments as for join hold for semi-join, outer-join, union, intersection, and difference.

---

[1] The standard example for aggregation is "sum of salaries by department." Department is called the grouping attribute here.

So far, we have demonstrated that reading and writing compressed data from and to disk is faster, and that it is not necessary to decompress data for the most frequent database operations. Let us consider some additional advantages of processing compressed data, and let us focus on joins.

First, materializing join output records is faster because records are shorter and less copying is required. Copying is a significant portion of CPU and bus activity in many database systems, and it is particularly important to reduce copying in shared-memory single-bus systems.

Second, for join inputs larger than memory, more records fit into memory. In hybrid hash join, for instance, the fraction of the file that can be retained in the hash table and thus be joined without any I/O is larger. During sorting for merge join, the number of records in memory and thus the number of records per run is larger, leading to fewer runs and possibly fewer merge levels.

Third, and very interestingly, skew is less likely to be a problem. The goal of compression is to represent the information with as few bits as possible. Therefore, each bit in the output of a good compression scheme has close to maximal information content, and bit columns seen over the entire file are unlikely to be skewed. Furthermore, bit columns will not be correlated. Thus, the compressed key values can be used to create a hash value distribution that is almost guaranteed to be uniform, i.e., optimal for hashing in memory and partitioning to overflow files as well as to multiple processors in parallel join algorithms.

To summarize this section, there are many uses for data compression in database management systems that have been ignored to-date. Compression not only improves I/O performance and hit rates at each "buffering" level in the storage hierarchy, it also can be exploited in database query processing without much change to implemented algorithms. We have outlined a number of performance effects of data compression; in the next section, we will quantify some of these effects in an example query.

## 4. Performance

For the purpose of this performance comparison, consider the I/O costs for hybrid hash join [7, 26] using 400 pages of memory of two relations R and S that require uncompressed 1,000 pages for R and 5,000 pages for S, or half as much compressed. For simplicity, we ignore fragmentation and assume uniform hash value distributions.

First, consider the cost using uncompressed data. The cost of reading the stored data is 6,000 I/Os. When building the hash table on R, 398 pages can remain in memory and 602 pages must be written to two build overflow files. During probing with input S, records equivalent to 1,990 pages can be joined immediately with the resident hash table, and 3,010 pages must be written to two overflow files. Next, the two pairs of overflow files must be joined, requiring 602+3,010 I/Os to read them. The entire cost is 6,000 I/Os on permanent files and 7,224 I/Os on temporary files, for a total of 13,224 I/Os.

Now consider joining compressed input relations. Reading the initial compressed data requires 500+2,500 I/Os. In the build phase, 399 pages remain in memory, 101 pages are written to an overflow file. In the probe phase, records equivalent to 1,995 pages are joined immediately,

and 505 pages are written to disk. Joining the two overflow files requires 101+505 I/Os. The entire cost is 3,000 I/Os to permanent files and 1,212 I/Os to temporary files, for a total of 4,212 I/Os.

The total I/O costs differ by a factor of more than three. While the I/O costs for the permanent files differ by a factor of two, as expected for a compression ratio of 50%, the I/O costs for temporary files differ by a factor of almost six. A factor of two could easily be expected; however, the improved utilization of memory (more records remain in the hash table during the build phase) significantly reduces the number of records that must be written to overflow files. Thus, compression reduces both the number and size of records written to temporary files, resulting in a reduction of I/O costs on temporary files by a factor of six.

If the compression scheme had been a little more effective, i.e., a factor of 2½ instead of 2 or a reduction to 40% instead of 50%, overflow files would have been avoided entirely for compressed data, leaving only the I/O on permanent data. The total I/O costs would have differed by a factor of 5½, 2,400 to 13,224 I/Os. Figure 4.1 shows the effect of the compression factor on hybrid-hash join performance for relations R and S. The numbers above the curve indicate the exact I/O cost for the compression factors marked at the bottom axis.

The graph can be divided into two regions. For compression factors below 2½, the build input is larger than memory, hash table overflow occurs, and I/O reduction by compression is more than the compression factor, similar to the example above. For compression factors above 2½, no overflow occurs and compression only reduces I/O on permanent files. It is very encouraging to observe in this graph that already very moderate compression factors, e.g., 1½, reduce the total I/O cost significantly. Even if some additional cost is incurred for decompressing output data, the performance gain through compressed permanent and temporary data on disk and in memory far outweights the costs of decompression.

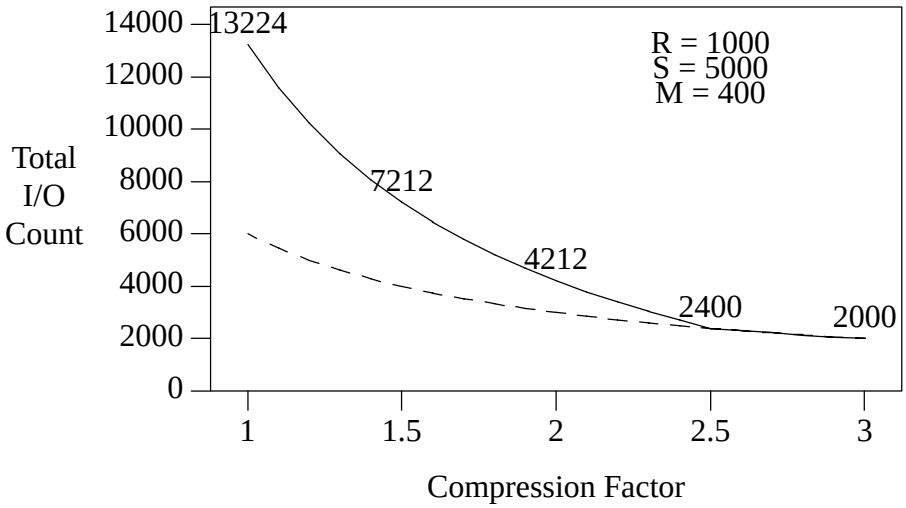


Figure 1. Effect of Compression on Hybrid Hash Join Performance.

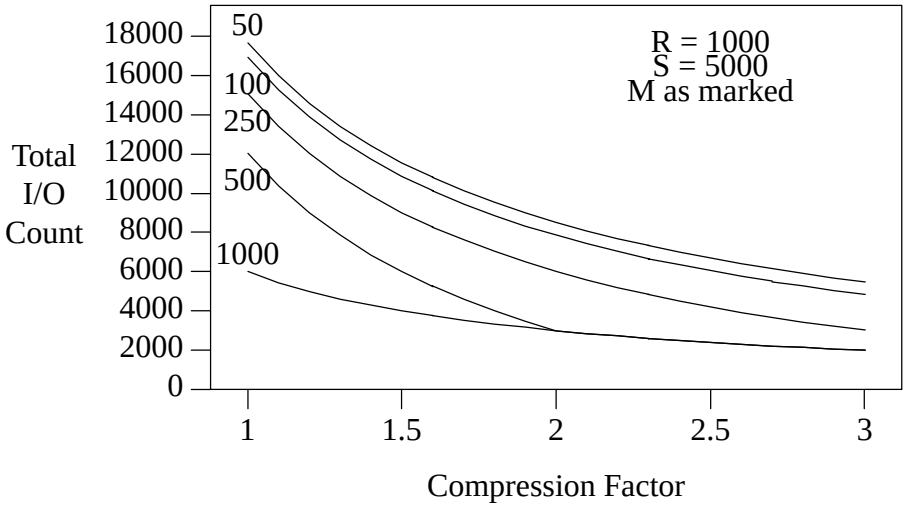


Figure 2. Effect of Compression and Memory Size on Hybrid Hash Join Performance.

Figure 4.2 shows the effect of compression on hybrid-join performance of R and S for a variety of memory sizes. The bottom-most curve for a memory size of 1,000 pages reflects the situation without overflow. The curve for 500 pages of memory has a steep gradient up to compression factor 2. Beyond this point, the hash table fits into memory and the curves for 500 and 1,000 pages coincide. For 250 pages of memory, which is ¼ of R, the curve joins the other curves without overflow at a compression factor of 4. For all smaller memory sizes, the hash table does not fit into memory in the considered of spectrum of compression factors. However, the performance gain is more than the compression factor for all memory sizes. For 50 or 100 pages of memory, the curves are very close to each other because almost all of R and S must be written to overflow files. If memory is very limited in a system, it might be more imporatant to allocate it to operators that may be able to run without overflow, and to use memory there with maximal efficiency, i.e., the best compression scheme possible.

Figure 4.3 shows the speedup for the previous figure. The bottom-most curve, for 1,000 pages of memory, represents linear speedup. All other curves indicate super-linear speedup. The curve for 500 pages has an obvious "knee" at compression factor 2 which had been already visible in the previous figure. For 250 pages of memory, the knee would be located at compression factor 4, at the edge of the graph, where the curve indicates a speedup factor of slightly more than 10. Considering that a speedup of 10 could be achieved with a compression factor of only 4 makes it imperative to further investigate the effect of compression on database query processing algorithms and their performance.

Figure 4.4 shows the performance for larger input relations. As in the previous figures, even a small amount of compression improves the performance significantly. However, for small memory sizes compared to the build input, the effect of fitting more compressed records than uncompressed records in memory is not as strong as for medium-size relations. Nonetheless, the gain in I/O performance is at least equal to the compression factor, and has the steepest decline

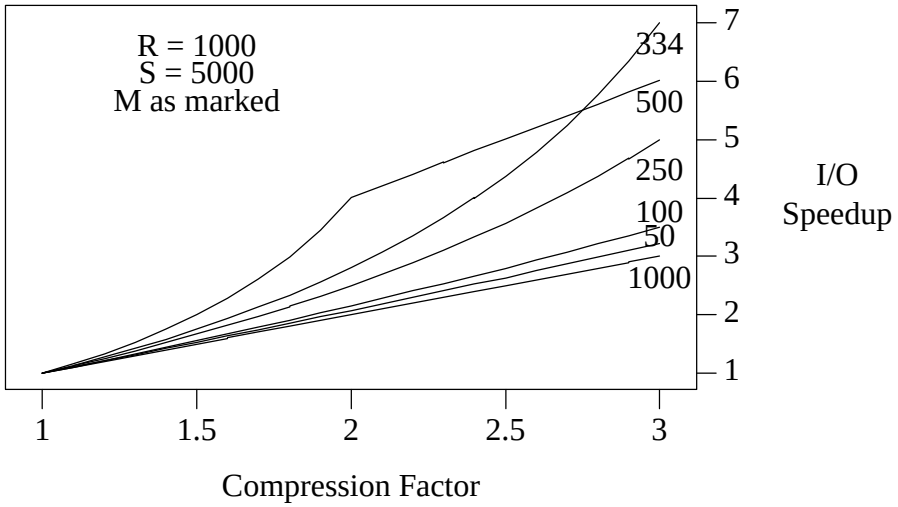


Figure 3. Speedup of Hybrid Hash Join through Compression.

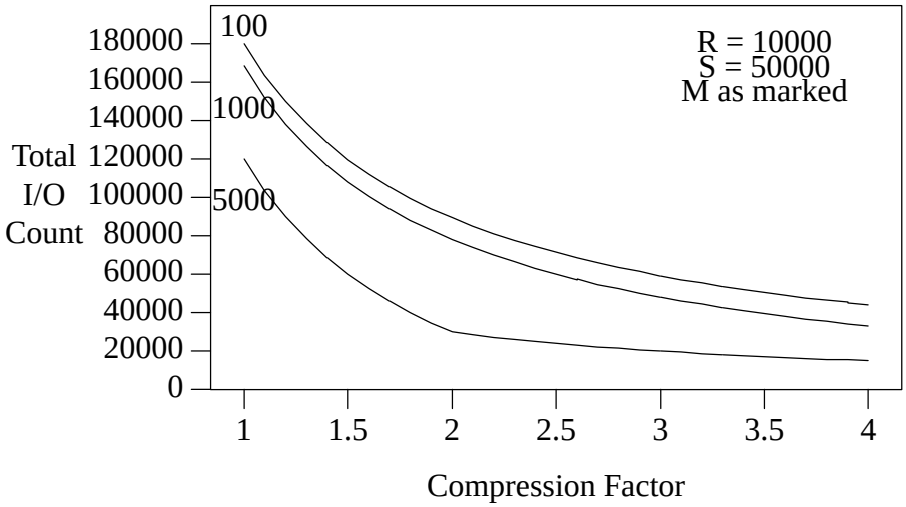


Figure 4. Effect of Compression on Large Hybrid Hash Join Performance.

for the small compression factors. For larger memory sizes, 5,000 pages in Figure 4.4, for which the build input size is a small multiple of the memory, we see the same effect as before — a steep gradient for small compression factors with a "knee" where the compression factor equals the quotient of build input size and memory size.

The same effects observed for a single hybrid-hash join can also be observed for complex queries and for sort-merge join. We omit this analysis and instead refer to the comparisons of sort-merge join and hybrid-hash join in [24, 26].

## 5. Summary and Conclusions

In this paper, we have outlined a set of fairly simple techniques to achieve database performance improvements by data compression. The key ideas are to compress attributes individually, to employ the same compression scheme for all attributes of a domain, and to perform data manipulations before decompressing the data. Not only do these techniques reduce space requirements on disk and I/O performance when measured in records per time for permanent and temporary data, they also reduce requirements of memory, thus reducing the number of buffer faults resulting in I/O. When data compression is used in conjunction with algorithms that use large work spaces, even modest compression can result in significant performance gains. Furthermore, our techniques for processing compressed data are very easy to implement.

In a simple performance comparison, we have seen that for data sets larger than memory performance gains larger than the compression factor can be obtained because a larger fraction of the data can be retained in the workspace allocated to a query processing operator. For a compression factor of two, we observed performance gains of factor three and more.

## Acknowledgements

This research has been partially supported by the Oregon Advanced Computing Institute (OACIS) and by NSF awards IRI-8805200, IRI-8912618, and IRI-9006348.

## References

[1] P. A. Alsberg, Space and Time Savings Through Large Data Base Compression and Dynamic Restructuring, *Proc. IEEE 63*, 8 (August 1975), 1114.

[2] T. Bell, I. H. Witten and J. G. Cleary, Modelling for Text Compression, *ACM Computing Surveys 21*, 4 (December 1989), 557.

[3] D. Bitton and J. Gray, Disk Shadowing, *Proc. Int'l. Conf. on Very Large Data Bases*, Los Angeles, CA, August 1988, 331.

[4] H. Boral and D. J. DeWitt, Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines, *Proc. Int'l. Workshop on Database Machines*, Munich, 1983, 166. Reprinted in A. R. Hurson, L. L. Miller, and S. H. Pakzad, Parallel Architectures for Database Systems, IEEE Computer Society Press, Washington, D.C., 1989.

[5] K. Bratbergsengen, Hashing Methods and Relational Algebra Operations, *Proc. Int'l. Conf. on Very Large Data Bases*, Singapore, August 1984, 323.

[6] G. V. Cormack, Data Compression In a Database System, *Comm. of the ACM 28*, 12 (December 1985), 1336.

[7] D. J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker and D. Wood, Implementation Techniques for Main Memory Database Systems, *Proc. ACM SIGMOD Conf.*, Boston, MA, June 1984, 1.

[8] S. J. Eggers and A. Shoshani, Efficient Access of Compressed Data, *Proc. Int'l. Conf. on Very Large Data Bases*, Montreal, Canada, October 1980, 205.

[9] S. J. Eggers, F. Olken and A. Shoshani, A Compression Technique for Large Statistical Data Bases, *Proc. Int'l. Conf. on Very Large Data Bases*, Cannes, France, September 1981, 424.

[10] L. Felician and A. Gentili, A nearly optimal Huffman technique in the microcomputer environment, *Inf. Sys. 12*, 4 (1987), 371.

[11] R. G. Gallager, Variations on a Theme by Huffman, *IEEE Trans. on Inf. Theory IT-24*, 6 (1978), 668.

[12] M. E. Gonzalez-Smith and J. A. Storer, Parallel Algorithms for Data Compression, *J. of the ACM 32*, 2 (April 1985), 344.

[13] P. Goyal, Coding methods for text string search on compressed databases, *Inf. Sys. 8*, 3 (1983), 231.

[14] G. Graefe and S. S. Thakkar, Tuning a Parallel Database Algorithm on a Shared-Memory Multiprocessor, *Software − Practice and Experience 22*, 7 (July 1992), 495.

[15] L. Haas, W. Chang, G. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey and E. Shekita, Starburst Mid-Flight: As the Dust Clears, *IEEE Trans. on Knowledge and Data Eng. 2*, 1 (March 1990), 143.

[16] D. A. Huffman, A method for the construction of minimum redundancy codes, *Proc. IRE 40*(1952), 1098.

[17] D. E. Knuth, Dynamic Huffman coding, *J. of Algorithms 6*(1985), 163.

[18] D. A. Lelewer and D. S. Hirschberg, Data Compression, *ACM Computing Surveys 19*, 3 (September 1987), 261.

[19] J. Li, D. Rotem and H. Wong, A New Compression Method with Fast Searching on Large Data Bases, *Proc. Int'l. Conf. on Very Large Data Bases*, Brighton, England, August 1987, 311.

[20] C. A. Lynch and E. B. Brownrigg, Application of Data Compression to a Large Bibliographic Data Base, *Proc. Int'l. Conf. on Very Large Data Bases*, Cannes, France, September 1981, 435.

[21] F. Olken and D. Rotem, Rearranging Data to Maximize the Efficiency of Compression, *J. of Computer and System Sciences 38*, 2 (1989), 405.

[22] M. Rodeh, V. R. Pratt and S. Even, Linear Algorithm for Data Compression via String Matching, *J. of the ACM 28*, 1 (January 1981), 16.

[23] S. S. Ruth and P. J. Keutzer, Data compression for business files, *Datamation 18*(September 1972), 62.

[24] D. Schneider and D. DeWitt, A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment, *Proc. ACM SIGMOD Conf.*, Portland, OR, May-June 1989, 110.

[25] D. G. Severance, A practitioner's guide to data base compression, *Inf. Sys. 8*, 1 (1983), 51.

[26] L. D. Shapiro, Join Processing in Database Systems with Large Main Memories, *ACM Trans. on Database Sys. 11*, 3 (September 1986), 239.

[27] J. A. Storer, *Data Compression: Methods and Theory*, Comp. Sci. Press, New York, NY, 1988.

[28] M. Toyama and S. Ura, Fixed Length Semiorder Preserving Code for Field Level Data File Compression, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, April 1984, 244.

[29] T. A. Welch, A Technique for High Performance Data Compression, *IEEE Computer 17*, 6 (June 1984), 8.

[30] R. N. Williams, Dynamic-historic predictive compression, *Inf. Sys. 13*, 1 (1988), 141.

[31] H. K. T. Wong and J. C. Li, Transposition Algorithms on Very Large Compressed Data, *Proc. Int'l. Conf. on Very Large Data Bases*, Kyoto, Japan, August 1986, 304.

[32] J. Ziv and A. Lempel, A Universal algorithm for sequential data compression, *IEEE Trans. on Inf. Theory IT-23*, 3 (May 1977), 337.

[33] J. Ziv and A. Lempel, Compression of individual sequences via variable-rate coding, *IEEE Trans. on Inf. Theory IT-24*, 5 (September 1978), 530.