Data Compression Support in Databases

Balakrishna R. Iyer
Database Technology Institute
IBM Santa Teresa Lab
San Jose, CA 95161-9023
balaiyer@vnet.ibm.com

David Wilhite
University of Southern California
Los Angeles, CA 90028
dwilhite@perspolis.usc.edu

Abstract

Computers running database management applications often manage large amounts of data. Typically, the price of the I/O subsystem is a considerable portion of the computing hardware. Fierce price competition demands every possible savings. Lossless data compression methods, when appropriately integrated with the dbms, yield significant savings. Roughly speaking, a slight increase in cpu cycles is more than offset by savings in I/O subsystem. Various design issues arise in the use of data compression in the dbms - from the choice of algorithm, statistics collection, hardware versus software based compression, location of the compression function in the overall computer system architecture, unit of compression, update in place, and the application of logic to compressed data. These are methodically examined and trade-offs discussed in the context of choices made for IBM's DB2 dbms product.

1 Introduction

Lossless compression methods are well known. The benefits to computer systems running database management system (dbms) products is particularly significant because a large portion of the computing system cost is attributed to the I/O subsystem. We conducted an informal survey of dbms users, and numerous instances were found where the price of the I/O subsystem exceeded the price of the processors. Amongst the users surveyed, we found they spent roughly equal amounts on the processors and the I/O subsystem. Their expectation was that, in the future, costs of the I/O subsystem would far exceed those of the processors. Most, if not all of their data was kept uncompressed. They had attempted compressing their data using application specific software compression but decided against it due to performance degradation in trial experiments. Another reason for rejecting compression was the increase in application complexity. How these objections were resolved will be described in the context of IBM's DB2 product.

After directing the reader to [GS91] and [Ba85] for an extensive discussion on the advantages of compression, we highlight the key benefit, disk savings. It will be described later in the paper that typical savings in disk range from 30 to 80% (of that needed to store uncompressed data). From a multitude of experiments conducted on data obtained from dbms users, we believe that 50% space savings is a reasonable expectation for compression. Again referring the reader to [Ba85] for an extensive discussion of the deterrents to the use of compression (each of which we address), we highlight the key inhibitor, cpu costs, in section 2.

The applicability of compression has not gone unnoticed. In [RK72] an argument is made for dbms to support compression using the Huffman encoding algorithm [Hu52]. Alsberg [Al75] studied various compression algorithms that can be viewed as pre-cursors to modern compression techniques. Alsberg argued both for several tables to share the same information for compression, and for independently compressing partitions of tables (both of which are features of DB2's implementation). Alsberg argued that compression makes rows smaller, hence more fit on a page, and this helps maintain clustering. Advantages of preserving collating sequence through compression, as it relates to clustering, appears in [TU84]. [LB81] contains a comprehensive analysis of compression based on technology of its time, raising many issues resolved in our work. Efficient search and retrieval of compressed data is discussed in [ES80]. A qualitative discussion of benefits and costs, some compression algo-

**Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994**

rithms, and applicability to business applications can be found in [Se83]. Earlier support of compression in IBM's IMS dbms product is described in [Co85]. Compression hardware for Huffman encoding is described in [RS91]. The argument for applying compression to columns of a row independent of one another, so they may be decoded independently, is advanced in [GS91] along with methods for applying relational operators to compressed data.

2 Database and Architectural Issues

2.1 Hardware vs. Software Compression

In this section, we will show the impact of compression on oltp applications using our best estimates of costs and DB2's implementation of compression as a model. We intend to show trends rather than make a specific prediction on system performance [GS91, RS91]. Based on studies of various implementations, we estimate software compression requires about 15 instructions per byte, while hardware compression requires only 1 instruction per byte. We estimate the TPC-B benchmark transaction [Gr91] to cost roughly 100K instructions. It reads and updates three rows of about 100 bytes each, and inserts a new 100 byte row. A total of 300 bytes must be decoded while 400 bytes are encoded. Software compression yields a total cost of 10.5K cpu instructions, or roughly a 10.5% increase in cpu cycles. In many cases this is well worth a 50% savings in disk space.

By their nature, decision support applications scan more data. As an example, we choose a table scan that scans all rows, evaluates predicates, and selects very few. Efficient dbms products can access the adjacent row in a table, locate columns and evaluate predicates in the low 100's of instructions per row[1]. For the sake of discussion, we assume 200 instructions per row. A typical 100 byte row will add 1500 instructions for software decoding, making it prohibitive for such use. Hardware decoding, on the other hand, adds only 100 instructions per row and is the implementation of choice for decision support applications.

Software compression is cost-effective in the narrow band of applications which are oltp intensive. Even a small amount of decision support makes software compression costs large[2]. On the other hand, hardware compression yields cost benefits over a broad range of oltp and decision support applications. It is for this reason that IBM's dbms products (DB2, IMS and VSAM) support hardware compression.

2.2 Compression Function Location

From a cost viewpoint, compression functions should be placed closest to the consumer/producer of data, at the point of capture and display, since all the hardware and software elements in the data flow will benefit. In fact, for video data types this is the only possible way current systems can sustain the flow rates needed for real-time display. Commercial data do not exhibit such high flow rates. In addition, both oltp and decision support workloads typically filter data. Only the account number, balance, and security code columns may be needed out of many columns in the account row to complete a transaction, say the withdrawal of $100. As explained later, it is not possible to locate a column of the compressed row (without unacceptable space overheads), hence decoding must occur before extracting these columns (a projection in relational dbms terminology).

In decision support applications, one may evaluate many predicates against each row accessed, and select only those rows that qualify. Even if we were able to locate the columns within a row, it is not known how to apply all the selection predicates we support in current dbms against compressed data (an interesting research opportunity). Hence, again the need to decode each row to apply selection logic. In current dbms products (with very few exceptions) functions of column extraction (projection) and applications of selection predicates lie in dbms code running on the processor. Hence it is necessary to decode the data by the time such logic is applied. This prohibits placing the compression functions only in the network adapters, for example. Of course, it is possible to exploit them during transmission by encoding the result of the data filtering done by the dbms.

Thus compression functions can be placed in the processor, the disk, or anywhere in between. If placed in the processor and used to decode rows only when they are manipulated by the application, and compressing them upon insertions, very few rows need to be kept decoded at any time. Almost all the data in memory can be kept compressed, yielding savings in all levels of the memory hierarchy up to the disk. If placed at the disk controller or disk, the benefit is reduced. For this reason IBM's DB2, IMS, and VSAM dbms products compress data with hardware provided by the processor (IBM's network communications product VTAM shares the compression hardware to compress data on the network).

2.3 Unit of Compression

Generally, if a collection of data is compressed together, then all bytes occurring before the needed byte have to be decoded on access. Oltp applications access just a few columns of one or a few rows. Decision support applications access a few columns of some or many rows of tables. Thus the ideal unit of compression is a column value as argued in [GS91]. Unfortunately, the vast majority of columns are fixed-length columns typically 4 to 6 bytes long. For uncompressed

[1] based on our measurements on commercial dbms products

[2] Software compression does reduce I/O response time. System costs are the sum of cpu and disk costs. Decision support contributes to a steep increase in cpu usage.

rows, DB2 tracks the length of a fixed-length column only once per table (in the metadata describing the table). If column values are individually compressed, each fixed-length column becomes a variable length column, and its length would need to be individually tracked for each row, costing a minimum of one byte per column per row. Assuming a 50% compression savings, the 4-6 byte columns that compress to 2-3 bytes will require 3-4 bytes of storage. Instead of getting 50% space savings, only a savings of 25 to 33% is realized. This limits the workload mixes for which compression saves in overall system price.

In IBM's DB2, IMS, and VSAM products, the next higher unit (the row - typically 40 to 120 bytes) was chosen as the unit of compression. Compressed rows are of variable length, thus the length of each row must be tracked. In the case of many existing dbms products, the row header already tracks the length of each row. This is needed even for fixed-length rows to support schema transformation of the addition of a column to a row (ALTER ADD COLUMN in SQL) efficiently. Thus, no extra length field is needed to to support variable length rows.

Compression is based on statistical properties of data. Updated rows may change in length. The performance of both oltp and decision support applications benefits from updated rows being stored back in place [IW94]. The needed space management algorithms to handle variable length rows already exist in most dbms. This provides another practical reason for implementing compression upon the dbms and the processor.

An additional detail is worth mentioning. It is necessary to consider the situation where an uncompressed row is slightly smaller than a page. Compressing the row may cause it to increase in size (because of the statistical basis of compression algorithms) so it no longer fits into a page. Dbms products have guaranteed that such a row will not be rejected. DB2's solution is to use a a bit in the row header to indicate whether the row is compressed. Another alternative is for the dbms to support splitting a row across a page like IBM's IMS dbms product.

2.4 Adaptive vs. Non-adaptive Compression

Compression methods can be characterized as the specification of data in terms of an assumed model of it. The model is usually parameterized by statistics gathered from the data to be compressed. If the parameters used by the model are not changed during the encoding or decoding of the data, the model is said to be non-adaptive, otherwise it is adaptive. The Unix compress call uses an adaptive compression algorithm.

Experiments with user data demonstrated that compression starting with no a priori knowledge using an adaptive model for compressing a row yields insignificant space savings. Rows are typically 40 to 120 bytes in length. Experiments suggest a row needs to be at least 512 bytes long for some of the popular adaptive compression methods to show significant savings. The phenomenon of poor start-up efficiency is documented in compression literature [BWC89], and there is no surprise that there is little opportunity for compression over a single row (40 to 120 bytes).

Over the length of the row, data types change every few bytes. Relationships identified over a decimal column do not apply when compressing a character column. This illustrates why adaptive compression without a priori knowledge does not work well when the unit of compression is a row. However, much can be learned when looking down a column, from row to row. This provides a source of a priori knowledge. The technique can be considered in terms of a two pass algorithm. In the first pass, available data is analyzed and statistics gathered to derive the parameter values to drive the model. These parameters are fixed and are used each time to begin compressing or decompressing a row. Within a row's compression (or decompression) the model can be adapted from the extra learning possible from the bytes in the row. Experiments suggest that little compression is to be gained from such adaption in this case. DB2 uses non-adaptive compression with a priori knowledge. However, while being non-adaptive over the length of a row, DB2's compression is nevertheless not static. It is possible to gather fresh statistics and re-parameterize the model during every orgranization.

2.5 Application of Logic to Compressed Data

The cpu intensiveness of compression leads inevitably to the search as in [GS91] for methods to store the data compressed and perform data manipulation without decompressing the data. The approach taken in [GS91] is to pick the column within a row as the unit of compression, then manipulations involving equality comparisons can be done on the compressed columns (e.g., hash join, selection via the "equality" predicate). Of significance to compression, we found B-tree based indices supported in almost every rdbms product. They improve performance for queries that have range selection predicates (e.g., SALARY $\leq$ 90K, SALARY BETWEEN 40K and 50K). Not surprisingly, a survey of DB2 user queries [TO91] showed 28% of user queries involve at least one BETWEEN predicate. Almost every query in the proposed TPC-D decision support benchmark [Ra93] involves range predicates. B-tree indices also provide a free sort for sort-merge joins, also found in many dbms products. Many of the navigations of the B-tree index (on compressed keys) can be performed without decoding the keys, if the compression algorithm preserves (sort) order. Each of the algorithms to be discussed in this paper can be modified to preserve sort order [HT71, GM59, Pa76, ZIL93]. A summary of each can be found in [IW94]. A good

topic for future research would be to compare, contrast, and evaluate the trade-off between the approach advocated in [GS91] and the use of order preserving compression.

2.6 Algorithm Alternatives

We examined various compression algorithms [IW94] and found two key ideas. One idea used in many algorithms was the recognition that different symbols occur with varying probability. These techniques assign shorter bit patterns to frequently occurring symbols at the cost of assigning longer bit patterns to less frequently used symbols. Huffman encoding [Hu52] and Arithmetic encoding [WNC87, BCW90, LR79, LR82, RM89, Pa76] are good examples. Decoding speed is an issue with these algorithms (naive decoding involves the executions of a few machine instructions for every coded bit), and is addressed in [Fr75]. Another idea used by compression algorithms is the identification of frequently occurring phrases of symbols. Frequently occurring phrases are stored in a dictionary. Encoding works by replacing a phrase (that appears in the dictionary) in a row by the label of its dictionary entry. The Ziv-Lempel compression algorithm and its variants [ZL77, ZL78, We84, MW85] are examples of compression algorithms that store a dictionary of frequently used phrases. The variant we use stores phrases in a parse tree.

3 Experimental Studies of Compression Techniques

3.1 Choice of Algorithm

Disk savings due to compression is dependent on the nature of data stored in databases. Image and video sources are known to produce data that can be reduced by a factor of 10 to 100. English text has been reported to reduce to 25-75% of its original size, depending on the choice of compression algorithm [BCW90]. The compressability of data stored in commercial dbms has not been widely reported.

We contacted users of IBM's IMS, DB2 and SQL/DS dbms products and requested randomly selected rows from their largest live production tables. Confidentiality of that data became a common issue. Some users were willing to give us their data after altering the confidential information. Evaluating the compressability of this modified data would not provide reliable results and such data was rejected. Approximately twelve customers provided tables that were unaltered along with their permission to use the tables over a specified time period. Because the periods did not completely overlap, we were not able to conduct all experiments over all the tables. Nevertheless, interesting results were found by using several tables for each experiment. One surprising observation was that data types were predominantly character, numeric, and time related. None of the data was image, audio, or video.

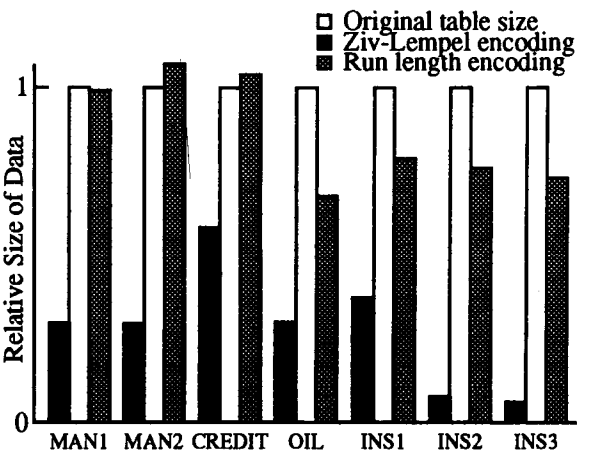


Figure 1: Comparison of Ziv-Lempel and RLE

Because the IMS dbms product already supports run length encoding (RLE), we first established the need for an alternative algorithm. We compared the RLE and Ziv-Lempel[3] algorithms over seven tables: 3 from DB2 users (MAN1 and MAN2 from a manufacturing application and CREDIT from a credit card authorization application), 1 from an SQL/DS user (OIL, an oil company application), and 3 from IMS users (INS1, INS2 and INS3, all insurance applications). The particular RLE algorithm studied used the first bit of an indicator byte to distinguish between runs of a repeated character and unencoded data. For runs of a repeated character, the remaining 7 bits of the indicator contained a repetition count. The indicator was followed by one occurrence of the repeated symbol. For unencoded data, the remaining 7 bits contained the length of the unencoded string following the indicator. Ziv-Lempel compressed the data to between 5 and 50% of its original size, as shown in Figure 1. RLE expanded the data in two cases. Mean savings due to Ziv-Lempel was 71.85% of the original table size, while only 13.65% for RLE. For IMS data, Ziv-Lempel yielded a savings of 83.08% compared to only 24.21% for RLE. It is possible that RLE may have been more valuable in early dbms products that did not efficiently support variable length columns. Variable length data would be stored in fixed-length columns by padding them with blanks, a perfect opportunity for RLE. Modern dbms products support variable-length columns efficiently; thus, RLE no longer provides good compression. RLE was dropped from consideration early in our study.

Next, Ziv-Lempel was compared to Huffman and arithmetic encoding. We ran a large number of experiments on many tables. Huffman and arithmetic encoding gave comparable compression savings, with

[3]For all studies in this paper where Ziv-Lempel parse tree size is not specified, trees consisted of 4K nodes.

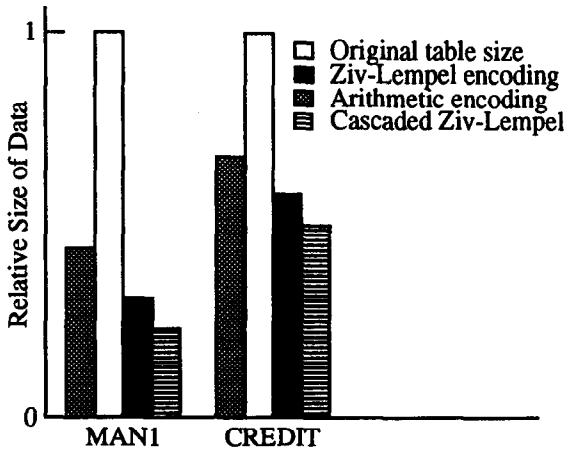


Figure 2: Comparison of Ziv-Lempel, arithmetic, and cascaded Ziv-Lempel compression

arithmetic encoding being marginally better. Savings due to arithmetic encoding will be discussed further. Arithmetic encoding based solely on the symbol occurrence probability of the 256 different bytes (a 0th order analysis) yields a savings from 30 to 50% of the original uncompressed size. Data structures needed for the algorithm must be stored in memory, and their cost in bytes is several times the number of symbols (256). For better compression, symbol probabilities can be conditioned on one or more previously occurring symbols. These 'higher order' models, however, require large data structures which are impractical to store in high speed processor cache (necessary for fast encoding and decoding). For this reason, a version of arithmetic encoding was chosen that used only a limited number of conditional probabilities (for the most frequently occurring symbols). As such, the results from arithmetic encoding should be viewed as a lower bound on possible savings. An active area of research in arithmetic encoding is the identification of encoding contexts that are most valuable with respect to compression savings and their efficient use. Furthermore, the size of high speed processor caches is larger for newer processors, making it practical to store more contexts and access them efficiently. Practical arithmetic encoding algorithms will most likely improve. Further discussion is beyond the scope of this paper. The third algorithm considered is a cascading of Ziv-Lempel and arithmetic encodings [PMK91]. Labels produced by Ziv-Lempel are themselves encoded (using arithmetic encoding) by exploiting the non-uniform distribution of occurrence of the labels. We call this the cascaded ZLA algorithm.

Data from many tables was analyzed. While the savings from each table varied, the relative ordering of the different algorithms based on compression savings surprisingly remained the same. Due in part to terms under which we obtained data from users, we are able to report results from only 2 tables. Fortunately, they represent the typical case, and the conclusions drawn from these tables are the same as those drawn from the larger set of tables analyzed. In Figure 2, we notice that both arithmetic and Ziv-Lempel encodings are reasonable choices for compression. Ziv-Lempel gives more compression than the particular arithmetic encoding algorithm evaluated. The gap is expected to be closed by future algorithms based on arithmetic encoding. Implementation issues are considered to differentiate the two algorithms.

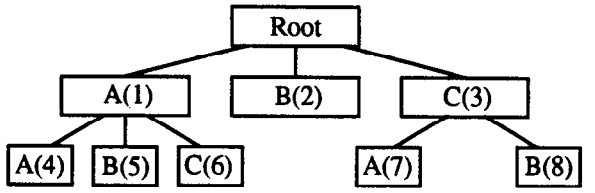
Folklore puts reads to be 3-4 times more frequent than writes for file systems. Due to the popularity of decision support tools, this ratio is likely to be more skewed towards reads for dbms. Thus implementation issues must focus on decoding. Decoding for non-adaptive Ziv-Lempel can be reduced to a table lookup using the tree label, retrieving the phrase represented by the label. Multiple bytes can be decoded in the time of a single memory access. Known arithmetic compression algorithms take more effort to decode single symbols. More research is needed to address this issue for arithmetic encoding implementations. For this reason alone, Ziv-Lempel was chosen over arithmetic encoding for DB2, IMS, and VSAM, and even over the cascaded ZLA algorithm which yielded more compression.

3.2 Evaluation of Ziv-Lempel Encoding

Since the cascaded ZLA algorithm gives more compression savings than the normal Ziv-Lempel algorithm, we were motivated to find ways to improve the compression savings from the normal algorithm without cascading. Before discussing variants of the algorithm, let us first examine basic parse tree construction.

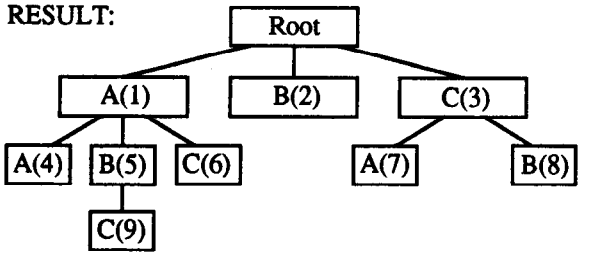
3.2.1 Parse Tree Construction and Sampling

Key to non-adaptive Ziv-Lempel compression is the construction of the parse tree for encoding and decoding. Several algorithms have been proposed for constructing Ziv-Lempel parse trees [ZL77, ZL78, We84, MW85]. The essential features of the basic parse tree building algorithm follows. Initially, the tree consists of a root node and its 256 children (each child node represents a distinct value of a symbol). The algorithm proceeds by scanning the input data, matching symbols against the tree, starting from the root (an example is described in Figure 3). Once a leaf node is encountered, a new node is added as a child of that leaf. The new node represents the next (unmatched) symbol of the input data. This is repeated until all data used for tree construction has been scanned. While encoding, the input data is scanned and matched (one symbol at a time) against the parse tree. When no further match is found, the label of the last node matched is output as the encoding for this phrase of symbols. This label, therefore, represents the concatenation of symbols found when traversing the tree from its root to this node.



Input Data: ABCABCBBB
Action: Match input data from root (match AB)
Action: Add next byte (add C)

RESULT:



Input Data Consumed: ABC
Input Data Remaining: ABCBBB
Action: Match input data from root (match ABC)
Action: Add next byte (add B)

Figure 3: Building a Ziv-Lempel parse tree

In building the parse tree, Ziv and Lempel [ZL77, ZL78] permit the tree to grow without bound while Welch [We84] and Miller and Wegman [MW85] refer to approaches that bound the size of the tree. In practice there is finite memory, hence only a finite number of nodes are available for the tree. Once the tree becomes full, a policy is needed for choosing a node for replacement. It should also be noted that the initial portion of the tree (the root and its 256 children) remains fixed. These nodes may not be chosen for replacement as they must exist in the final parse tree (this restriction permits the encoding of any symbol that may occur during encoding, even if it never appeared during parse tree construction). Several tree pruning algorithms were considered based on least frequency or recency of use. The results were insensitive to these choices.

The parse tree needs to be built when the table is first loaded. When the table to be loaded is available in off-line media (tape) or on-line media (disk), it is possible to make a pass through it for analysis. If the data arrives "one row at a time" at the database over an extended period of time, it is not practical to wait until all this data has arrived. If one assumes that the arrivals are not correlated with the contents of the rows, each arrival provides a good random sample. Although such is always not the case, we may still use the earliest arriving rows to a freshly created database as samples from which to construct the parse tree. However, during reorganization we can sample rows from the entire table. This may result in higher compression savings after the first reorganization. Early user feedback suggests the improvement to be in the 0-5% range.

The earliest arriving rows may be compressed only after the parse tree is constructed. A single row (typically in size from 40 to 120 bytes) is unlikely to contain enough information to build a parse tree with 4K nodes. It is possible that rows can be prevented from being loaded until sufficient number arrive to construct the parse tree. However, this is a poor idea if the required number of rows do not arrive within a short time interval. It is better to load the rows used to construct the parse tree immediately upon their arrival. They are loaded uncompressed. Once sufficient rows have arrived and the parse tree has been constructed, the loaded rows may be re-accessed and compressed. If the number of rows so affected is a small portion of the number of rows that populate the table, leaving these rows uncompressed has an insignificant effect on savings. This tactic is adopted by DB2, exploiting again the bit in the row header for indicating an uncompressed row.

During reorganization, techniques are needed for choosing the number of rows to sample and which rows to sample. By varying the number of randomly sampled rows used to build the tree, we studied the impact of the number of samples on compression savings.

Figure 4 shows the relative size of the compressed table to the uncompressed table as more rows were randomly sampled for constructing the parse tree. It is clear that if only a few percent of the rows were sampled, a parse tree can be built that compresses as well as a parse tree built with sampling more rows. Note that the tables we obtained from users are already samples of larger production tables in use. Interestingly, compression increases with the number of samples taken, but only up to a point. There is a levelling off, and almost a small loss afterwards. At some threshold, sampling more rows no longer improves compression. Figure 4 also shows the points at which the number of samples taken is just enough to construct a 4K node parse tree (marked with an 'o'). These points occur at or near the threshold at which compression is maximized. Our experiments suggest that the number of rows needed to fill the parse tree is a good indicator of the number of rows which should be sampled to attain good compression.

If sampling is performed, the question arises as to which rows to sample. Ideally, randomly chosen samples should be used to construct the parse tree. However, random sampling can be quite expensive, since accessing each sample from a very large table may involve an I/O. Furthermore, the random sampling of a table stored on tape is prohibitive. However, most reorganization algorithms have an "unload" phase, during which all rows are scanned. It is desirable to pick samples during this single scan of the table. Simple algorithms like picking every n^{th} row fail to work be-

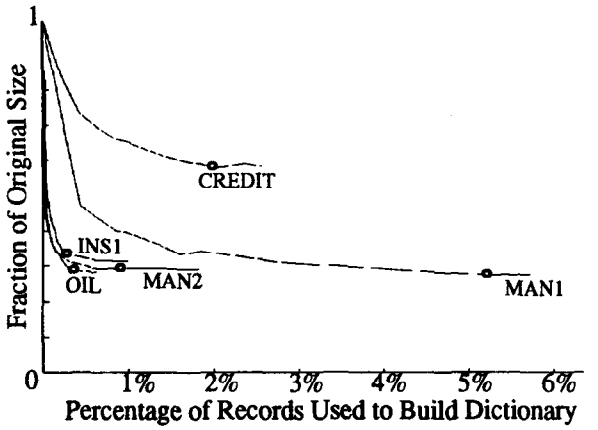


Figure 4: Use sampling to build Ziv-Lempel parse tree

cause available row counts for the table may be unreliable, and the number of rows needed to fill the parse tree is not known before the scan. In this context, sampling algorithms have been discussed in [OR86] and [Vit85]. As per these methods, samples are drawn and placed in a reservoir. The methods manage the placement and replacement of samples from the reservoir and their final use. The most relevant reference is [ASW85], where an incremental sampling algorithm attributed to Wegman is described (and referred to as "sample counting") for another application. The algorithm begins with fine granularity sampling (i.e., every row) and gradually makes the sampling granularity coarser (i.e., every other row, every fourth row, etc.). However, the method to control the change in sampling granularity is quite different since the application is different.

DB2's sampling during reorganization works similarly. Every row is sampled from the beginning of the table until the parse tree is full. Sampling granularity is made coarser until the same number of rows have been sampled. At this point, sampling granularity is made even more coarse, and the algorithm iterates until the entire table has been scanned.

3.2.2 Parse Tree Size

Next, the size of the parse tree was chosen. Tree sizes from 512 nodes to 16K nodes were implemented. Note that trees with more nodes have longer labels. 512 nodes can be represented with 9 bits, while 4K nodes need 12 bits to distinguish them. Experimental results from using 512 to 16K node parse trees are given in Figure 5. Increasing the number of tree nodes increased compression savings. A significant increase in compression savings is shown from 512 to 1K tree nodes. The increase in compression leveled off at about 4K nodes. Since it is desireable for the parse tree to fit in high-speed cache, DB2 uses 4K nodes as the default size for the parse tree.

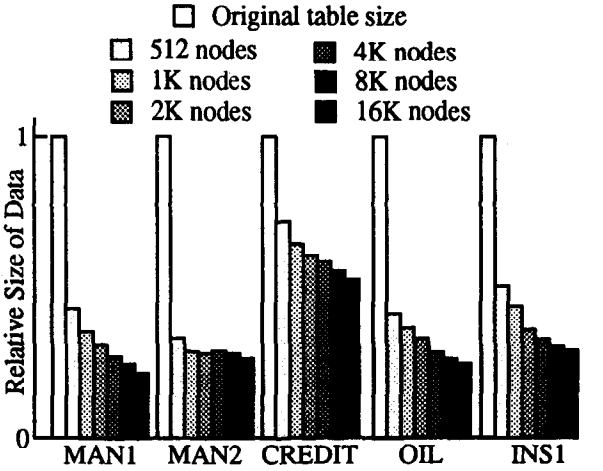


Figure 5: Ziv-Lempel compression tree sizes

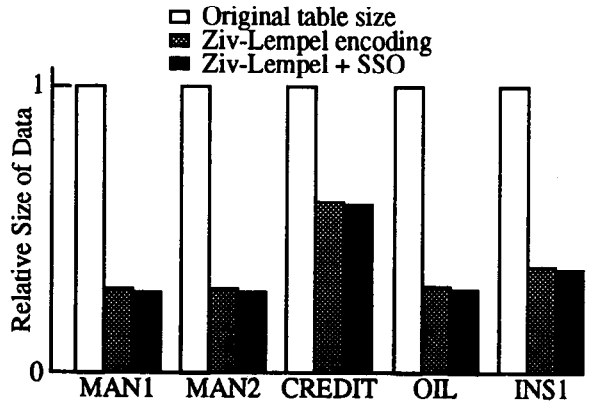
3.2.3 The Short Symbol Option and Extension Symbols

With a 4K node tree, 12 bits are used per label. Every label output for a level 1 node loses 4 bits of compression (12 bits are used to represent an 8 bit symbol). A particular variant of the Ziv-Lempel algorithm due to Plambeck [Pl89] was evaluated to minimize this loss. The idea of this variant is to let labels of the first level nodes be represented by the value of the symbol, while the remaining labels (to deeper level nodes) are represented by the original 12 bit values. To distinguish short symbols from regular symbols, an extra bit is added to each symbol. Figure 6a presents the compression savings with and without this variation. The algorithm with this variation is referred to as SSO (for short symbol option). The average compression savings gained was about 1%, and this approach was abandoned.

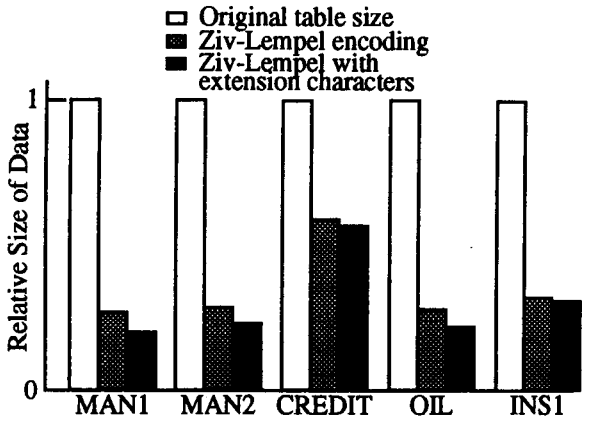
Another variation of Ziv-Lempel due to [MW85] was investigated. In this variant, nodes in the tree (other than level 1 nodes) represent **extended symbols**. An extended symbol represents a phrase of symbols, rather than a single symbol. During tree building, whenever a node is added to the tree, it represents the phrase matched subsequent to the current match, rather than simply the next symbol following the current match. The average amount of compression gained was over 4% of the original table sizes, as shown in Figure 6b. This enhancement is used by DB2.

3.2.4 Parse Tree Trimming

Some of the experiments reported in Figure 4 show that sampling more rows than needed to build a parse tree could reduce the amount of compression savings. A possible reason is that the algorithms used to replace tree nodes (when the tree becomes full) are inadequate. Our algorithms replace one node whenever an entry must be added to the tree after it has reached



(a) Short Symbol Option



(b) Extension Symbols

Figure 6: Ziv-Lempel algorithm variations

its maximum size (4K nodes). Since this decision is based on looking at only 1 additional (possibly extended) symbol, perhaps it may be improved. In the alternative explored, the parse tree was permitted to grow beyond its final size, and pruned after final construction. For example, if a 4K node parse tree is built, it was allowed to grow to 8K nodes during its construction. Once the tree reached 8K nodes, the single node replacement algorithm described earlier was used. The frequency of reference to each node was also stored during tree building. After the parse tree construction phase, the tree contained 8K nodes. A simple tree trimming algorithm was used which identified the nodes with fewest references and trimmed them (note that only leaves were trimmed) so the parse tree contained 4K nodes. We conducted a number of experiments, increasing the built tree size to 2, 4, 8, and 16 times the size of the desired parse tree, then trimmed it. We found that compression savings increased to an optimum when the multiplicative factor was 4. When the built tree was larger than a factor of 4, we saw loss of compression savings consistently for each of the different tables. The improved savings is not large, as shown in Figure 7 (on the average, slightly more than

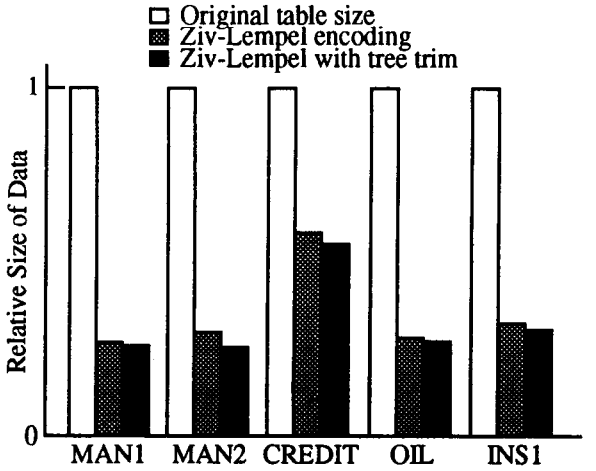


Figure 7: Build 16K node tree, trim to 4K nodes

2%). However, since the implementation cost is low and it only affects the tree construction phase, it is included in DB2.

3.2.5 Column Sensitivity

Rows are comprised of multiple columns. During parse tree construction, as each new row is parsed, parsing resets to the root of the tree. Furthermore, whenever a phrase from the input row is parsed against the current tree, parsing resets to the root of the tree. Column sensitivity experiments were conducted which reset parsing to the root of the tree at various column boundaries. For our first column sensitivity experiment, we reset parsing to the root of the tree at every column boundary during parse tree construction. During the actual encoding of data, since speed is an issue, encoding was not sensitive to column boundaries. In summary, the algorithm was column sensitive (to every column) during the parse tree construction phase and column insensitive during the encoding phase. This naive approach proved to be disastrous as shown in Figure 8. For example, MAN2 suffered severe compression loss, due to many small columns in the table. Intuitively, however, it seems that selectively choosing certain column boundaries may provide better compression. The best results are presented from many related experiments. Resetting the parsing at some column boundaries did provide some improvement in compression (Figure 8). However, improvement was not significant enough to recommend this method.

The second experiment was similar to the first, except that the parse was reset at certain columns during both the build and encode phases. Results from this experiment were similar to those in the first experiment, and this method is not recommended.

The third experiment on column sensitivity used data of different types to build different parse trees (one for each type). A total of 4K nodes was used. The nodes were distributed among the trees in proportion to the space used by its corresponding type in

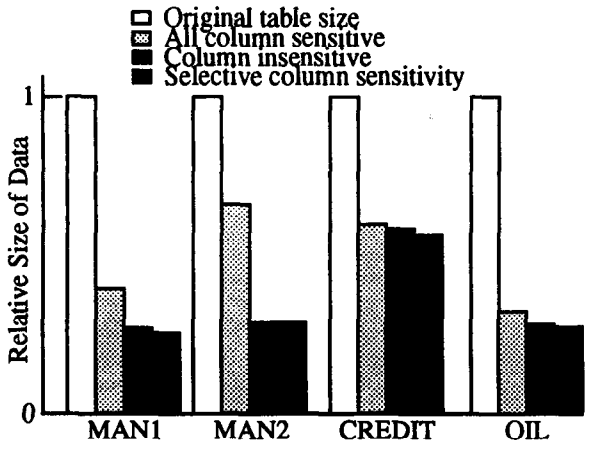


Figure 8: Column sensitivity, build phase only

the table. For example, if 46% of each row consists of character data, then the tree built for character data contains 46% of the 4K nodes. The results show that using a total of 4K nodes and compressing each column type separately provides approximately equivalent compression to the column insensitive method.

In summary, the results of the column sensitivity experiments were inconclusive. While suggesting that there are areas to explore for improving compression savings, no clear method emerged for exploiting column sensitivity. Further study is recommended.

4 Usability and Performance

The following DB2 features make compression easy to use. It is easy to enable and disable. Simply by using the ALTER command, DB2 is told to use compression. In the next reorganization (or if the table is empty, during the initial load of data) compression goes into effect. In addition, the maximum number of rows that can reside in a page is also doubled, in anticipation of more compressed rows stored per page.

It is possible to selectively apply compression. Frequently accessed and small tables may not need compression. Large but less used tables can be selectively compressed. DB2 also supports the partitioning of a table by key range and independent utility operations against various partitions. This allows DB2 to selectively compress partitions of a table. For example, time-series applications (such as retail and billing applications), frequently access data from recent months. The partition containing the most recent months can be left uncompressed while the remaining partitions are compressed. To help the user, tools are available which estimate the compression savings.

It can be argued that for frequently reorganized databases, the statistical properties with respect to compression do not change between each pair of consecutive reorganizations. Thus DB2 allows an existing parse tree to be reused during reorganization.

Although IBM's dbms products implement hardware compression, software compression is also supported. Data compressed by hardware is decodable by software and vice-versa. This allows the migration of applications and dynamic load balancing among multiple processors in an installation where only some of the processors have hardware compression support.

In the past, dbms users of software compression have reported a total loss of data if the software compression routine was lost by error. This is a problem solved by DB2. DB2's compression parse tree is regarded as metadata and stored in the header pages of a table. The parse tree is logged on creation and change, and is recoverable. When a table is migrated from one DB2 dbms to another, the parse tree goes along (the table is self-describing). Feedback from early users of DB2 compression is extremely positive.

When considering performance, oltp workloads gained significant price performance benefits from compression. Decision support workloads also benefitted. Table scans have been benchmarked to show a 50% reduction in response time at the cost of 20% extra CPU. Database backup and recovery have measured a 50% reduction in response time and a 10% reduction in CPU time. Utilizing an existing dictionary during database reorganization has yielded a 50% savings in reorganization time. Furthermore, it should be noted that the percentage cost of the dictionary build during reorganization decreases with table size due to the incremental sampling technique used.

5 Conclusions

We have made the case for integrating lossless compression into dbms products as a means of improving price performance. Tradeoffs were made to solve numerous architectural problems encountered during this integration. We chose the row as the unit of compression. The compression algorithm selected is a non-adaptive variant of the Ziv-Lempel algorithm using extension symbols. The Ziv-Lempel parse tree is determined by building a large parse tree with sampled rows and trimming it to the desired number of nodes. Column sensitivity was not employed because experiments with real data were inconclusive (while showing good potential). We have described the particular usability features and performance of this implementation for the DB2 dbms product. In conclusion, we believe that a majority of concerns relating to the use of compression for databases have been addressed, and expect wide acceptance of compression in dbms products.

6 Acknowledgements

We would like to thank many IBM employees and users of IBM's dbms products for their generous help. A partial list follows: Ron Arps, John Babb, Rick Baum, Nancy Burchfield, Albert Chang, Chung Chang, Walter Chang, Jo Cheng, Ivy Wong Chong, Carissa Chun, Dick Crus, Jean-Jacques Daudenarde, Greg

Davoll, Paramesh Desai, Mohamed El-Ruby, Joel Farber, Craig Friske, Debbie Fuhrer, Don Haderle, Harold Hall, Dave Hauser, Suse Kelley, Lubor Kollar, Gopal Krishna, Spencer Krueger, Clark Kurtz, Glen Langdon, Pete Lazarus, Ron Lember, Helen McMillan, Ted Messinger, Victor Miller, Rich Pasco, Ken Plambeck, Guru Rao, Pat Selinger, Akira Shibamiya, Bhaskar Sinha, Mark Slovick, Brian Smith, Barry Stevenson, Dave Voss, Robin Williams, John Wong, Pong Wong and Ahmad Zandi.

References

[Al75] P. Alsberg. Space and Time Savings Through Large Data Base Compression and Dynamic Restructuring. *Proc. IEEE*, 63(8), Aug. 1975, p 1114.

[ASW85] M. Astrahan, A. Schkolnick, and K. Whang. Counting Unique Values of an Attribute without Sorting. IBM Research Report RJ4960, Dec. 30, 1985.

[Ba85] M. Bassiouni. Data Compression in Scientific and Statistical Databases. *IEEE Trans. on Software Eng.*, SE-11 (10), Oct. 1985, p 1047.

[BCW90] T. Bell, J. Cleary, and I. Witten. Text Compression. Chapter 1, Prentice Hall, Englewood Cliffs, NJ, 1990.

[BWC89] T. Bell, I. Witten, and J. Cleary. Modelling for Text Compression. *ACM Computing Surveys*, 21, 4, Dec. 1989, p 557.

[Co85] G. Cormack. Data Compression in a Database System. *Communications of the ACM*, 28, 12, Dec. 1985, p 1336.

[ES80] S. Eggers and A. Shoshani. Efficient Access of Compressed Data Performance. *Proc. of VLDB*, Montreal, Oct. 1980, p 205.

[Fr75] A. Frank. Uniform Decoding of Minimum Redundancy Codes. U.S. Patent 388347, 1975.

[GM59] E. Gilbert and E. Moore. Variable-Length Binary Encodings. Bell Systems Technical Journal, 38, 1959, p 933.

[Gr91] J. Gray. The Performance Handbook for Database and Transaction Processing Systems. Morgan Kaufman, San Mateo, 1991.

[GS91] G. Graefe and L. Shapiro. Data Compression and Database Performance. *Proc. of ACM/IEEE Computer Science Symp. on Applied Computing*, Kansas City, Apr. 1991.

[HT71] T. Hu and A. Tucker. Optimum Binary Search Trees. SIAM J. Applied Math, 21(4), 1971, p 514.

[Hu52] D. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proc. of the IRE*, 40, Sept. 1952, p 1098.

[IW94] B. Iyer and D. Wilhite. Data Compression: A Method to Improve Price Performance for Databases. Technical Report, under preparation.

[LB81] L. Lynch and E. Brownrigg. Application of Data Compression to a Large Bibliographic Data Base. *VLDB*, France, Sept. 1981, p 435.

[LR79] G. Langdon Jr. and J. Rissanen. Arithmetic Coding with Integer Code Word Lengths. IBM Research Division Report RJ2597, San Jose, CA, Aug. 1979.

[LR82] G. Langdon Jr. and J. Rissanen. A Simple General Binary Source Code. IEEE Trans. on Information Theory, v. IT-28, Sept. 1982, p 800.

[MW85] V. Miller and M. Wegman. Variations on a Theme by Lempel and Ziv. Combinatorial Algorithms on Word, Spriger Verlag (A. Apostolico and Z. Galil, eds.), p 131.

[OR86] F. Olken and D. Rotem. Simple Random Sampling from Relational Databases. *Proc. of VLDB*, Kyoto, Japan, Aug. 1986, p 160.

[Pa76] R. Pasco. Source Coding Algorithms for Fast Data Compression. Ph.D. Thesis, Dept. of Electrical Engineering, Stanford University, 1976.

[PMK91] Y. Perl, V. Maram, and N. Kadakuntle. The Cascading of the LZW Compression Algorithm with Arithmetic Coding. *Proc. of IEEE Data Compression Conf.*, Utah, 1991, p 277.

[Pl89] K. Plambeck. Private Communications. 1989.

[Ra93] F. Raab. TPC-D Benchmark (Decision Support), Working Draft 5.1, June, 1993.

[RK72] S. Ruth and P. Keutzer. Database Compression for Business Files. *Datamation*, 18, Sept. 1972, p 62.

[RM89] J. Rissanen and K. Mohiuddin. IEEE Trans. on Communications, 37(2), Feb. 1989, p 93.

[RS91] N. Ranganathan and H. Srinidhi. A Suggestion for Performance Improvement in a Relational Database Machine. Computers Electrical Engineering, 17(4), 1991, p 245.

[RV93] M. Roth and S. Van Horn. Database Compression. *SIGMOD Record*, 22(3), Sept. 1993.

[Se83] D. Severence. A Practitioner's Guide to Database Compression. *Information Systems*, 8(1), Jan. 1983, p 51.

[TO91] A. Tsang and M. Olschanowsky. The Study of Database 2 Customer Queries. TR03.413, IBM Santa Teresa Lab, San Jose, CA, April 1991.

[TU84] M. Toyama and S. Ura. Fixed Length Semi-Order Preserving Code for Field Level Data File Compression. *Proc. of IEEE Data Engineering*, Los Angeles, Apr. 1984, p 224.

[Vit85] J. Vitter. Random Sampling with a Reservoir. *ACM Trans. on Math. Software*, 11(1), Mar. 1985, p 37.

[We84] T. Welch. A Technique for High Performance Data Compression. *IEEE Computer*, 17(6), June 1984, p 8.

[WNC87] I. Witten, R. Neal, and J. Cleary. Arithmetic Coding for Data Compression. *Comm. of the ACM*, 30(6), June 1987, p 520.

[ZIL93] A. Zandi, B. Iyer, and G. Langdon. Sort Order Preserving Data Compression for Extended Alphabets. *DCC 93 Data Compression Conf.*, 1993, p 330.

[ZL77] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Trans. on Information Theory*, 23(3), May 1977, p 337.

[ZL78] J. Ziv and A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Trans. on Information Theory*, 24(5), Sept. 1978, p 530.