Compressing Relations and Indexes

Jonathan Goldstein **Raghu Ramakrishnan**
Uri Shaft
Department of Computer Sciences, University of Wisconsin–Madison
1210 W. Dayton St., Madison, Wisconsin 53706
Tel: (608)262-6611, Fax: (608)262-9777
email: `(jgoldst, raghu, uri)@cs.wisc.edu`

February 18, 1998

Abstract

We propose a new compression algorithm that is tailored to database applications. It can be applied to a collection of records, and is especially effective for records with many low to medium cardinality fields. In addition, this new technique supports very fast decompression. (Gzip typically takes 10 times the CPU time to decompress relations, while achieving about the same compression ratio.) Our algorithm achieves compression on numeric fields only; while low and medium cardinality fields of other types can be handled by mapping values to a small range of integers, our algorithms must be complemented by other compression techniques for general non-numeric (text, image) fields. Promising application domains include decision support systems (DSS), since 'fact tables', which are by far the largest tables in these applications, contain many low and medium cardinality fields and typically no text fields [Kim96]. Further, our decompression rates are faster than typical disk throughputs for sequential scans; in contrast, gzip is slower. This is important in DSS applications, which often scan large ranges of records.

An important distinguishing characteristic of our algorithm, in contrast to compression algorithms proposed earlier, is that we can decompress individual tuples (even individual fields), rather than a full page at a time. This offers significant improvements in cases where a page is stored in the buffer pool and repeatedly probed for a few tuples; the page can be stored in compressed form, and retrieving an individual tuple is extremely fast (since there is no I/O, the CPU cost of decompression dominates here).

Our compression algorithm also improves index structures such as B-trees and R-trees significantly by reducing the number of leaf pages and compressing index entries, which greatly increases the fan-out. In this context, an important feature of our algorithm is that compression can be either lossy or lossless, where either the degree of loss, or the compression ratio, is user-specified. If data entries in the index are simply represented as $\langle key, rid \rangle$ pairs, using our compression technique naturally yields the benefits of such optimizations as storing $\langle key, rid - list \rangle$ pairs and recognizing "runs" of rids.

1 Introduction

Traditional compression algorithms such as Lempel-Ziv [Sto88, LZ76, LZ77], which is the basis of the standard gzip compression package, require uncompressing a large portion of the file even if only a small part of that file is required. For example, if a relation containing employee records is compressed page-at-a-time, as in some current DBMS products, a page's worth of data must be uncompressed to retrieve a single tuple. Page-at-a-time compression also leads to compressed 'pages' of varying length that must be somehow packed onto physical pages, and the mapping between the original pages/records and the physical pages containing compressed versions must be maintained. In addition, compression techniques that cannot decompress individual tuples on a page store the page decompressed in memory, leading to poorer utilization of the buffer pool (in comparison to storing compressed pages).

We present a compression algorithm that overcomes these problems. The algorithm is simple, and can be easily added to the file management layer of a DBMS since it supports the usual technique of identifying a record by a $\langle$*pageid, slotid*$\rangle$ pair, and requires only localized changes to existing DBMS code. Higher layers of the DBMS code are insulated from the details of the compression technique. In addition, this new technique supports very fast decompression of a page, and even faster decompression of individual tuples on a page. Our contributions are:

Page level Compression. We describe a compression algorithm for collections of records (and index entries) that can essentially be viewed as a new page-level layout for collections of records (Section 2). It allows decompression at the level of a specified field of a particular tuple; all other proposed compression techniques that we are aware of require decompressing an entire page. Scenarios that illustrate the importance of tuple-level decompression are presented in Section 4.4.

Performance Study. We present a performance analysis that underscores the importance of compression in the database context (Section 4). Our numbers are based upon a complete implementation of the algorithms presented in this paper. We measure the compression ratios and compression/decompression speeds achieved by our technique on a range of synthetic and real datasets, and compare with gzip, applied on a page-at-a-time basis. (Current systems, in particular Sybase IQ, [1] use a proprietary variant of gzip, applied page-at-a-time.) We typically get compression ratios of 3 or 4 to 1. On low cardinality datasets, we see compression ratios go as high as 88 to 1. These numbers are comparable to gzip, except that for low-cardinality data, our compression is better than gzip because of how gzip is applied page-at-a-time (the compressed data is rounded up to blocks of a minimum size, sacrificing some of the compression). Our algorithm is typically 10 times faster than gzip in decompressing a full page, and orders of magnitude faster if only an individual tuple is required. If all tuples on a page are required, our decompression is about 3 times faster than sequential I/O; in contrast, gzip is 2 to 7 times slower than sequential I/O.

Application to B-trees and R-trees. We study the application of our technique to internal nodes of index structures (e.g., B-trees and R-trees) in Section 3. We treat the $\langle key, pointer\rangle$ entries in an index as data and compress both the *key* and the *pointer* fields. In comparison to traditional techniques like storing $\langle key, rid-list\rangle$ pairs, and using special rid representations that exploit "runs" of rids, we achieve compression that should be comparable, with no additional effort (beyond compressing pages using our algorithm). Further, for R-trees we can choose between lossy and lossless compression for the *key* part of the data, in a way that exploits the semantics of an R-tree entry; a capability that is not possible with other compression algorithms. The *key* represents a hyper-rectangle, and in lossy compression we can use a larger hyper-rectangle and represent it in a smaller space.

Multidimensional bulk loading algorithm. We can exploit a sort order over the data to gain better compression. B-tree and R-tree orders are both utilized well. If data is initially unsorted, we present a bulk loading algorithm that has the following qualities:

- The algorithm first sorts the data packs the data into pages. These pages are a very high quality leaf level of an R-tree. Note that these pages can be used as a compressed version of the data without creating the rest of the tree.

[1] We thank Clark French at Sybase IQ for giving us information about the use of compression in Sybase IQ.

- A slightly modified version of the algorithm can ensure that there is no overlap between leaf pages of the tree for point data.

The compressed file produced by our algorithm is the set of leaf-level pages of an indexing structure (either an R-tree or a B-tree depending on the sorting algorithm used); retaining the entire tree increases the size of the compressed file by no more than 10% typically. Thus, the total size of the compressed file plus a clustered index with a, potentially multidimensional, search key is much less than the size of the original file! In addition to the compression, therefore, we obtain an index for little additional cost.

2 Compressing a Relation

The first part of our algorithm, which we call page level compression, takes advantage of common information amongst tuples on a page. This common information is called the *frame of reference* for the page. Using this frame of reference, each field of each tuple can be compressed, sometimes quite significantly; thus many more tuples can be stored on a page using this technique than would be possible otherwise. The compression is done incrementally while tuples are being stored, either at bulk-loading time or during run-time inserts of individual tuples. This ensures that additional tuples can be fit onto a page, taking advantage of the space freed by compression.

As in a conventional page organization, the information specific to a given tuple is stored in a 'slot' on the page, and the pageid and slotid together identify a tuple. To retrieve a tuple, we can retrieve the information in the corresponding slot, and in conjunction with the frame of reference information for the page, reconstruct the tuple (actually this can even be done for individual fields within a tuple). An important feature of our technique is that when fixed length fields are compressed, they remain fixed length. Thus our compression technique doesn't force the use of a slot directory. Note that all information needed to uncompress a page is stored on the same page.

The degree of compression obtained by our page-level compression technique depends greatly on the range of values in each field for the set of tuples stored on a page. Thus, the effectiveness of the compression can be increased, often dramatically, by partitioning the tuples in a file across pages in an intelligent way. If a database contains 500,000 tuples, say, there are many ways to group these tuples, and different groupings may yield drastically different compression ratios. [NR95] demonstrates the effectiveness of using a B-tree sort order to assign tuples to pages. We further develop the connection between index sort orders, including multidimensional indexes like R-trees, and improved compression.

We describe the page-level compression technique in Section 2.1, and the bulk-loading techniques for grouping records into pages in Section 2.2.

2.1 Page-Level Compression: Frames of Reference

Our basic observation is as follows: if we consider the actual range of values that appear in a given column on a given page, this is much smaller than the range of values in the underlying domain. For example, if the first column contains integers, the smallest value on the page in this column is 33 and the largest is 37, the range (33, 37) is much smaller than the range of integers that can be represented (without overflow). If we know the range of potential values, we can represent any value in this range by storing just enough bits to distinguish between the values in this range. In our example, if we remember that only values in the range (33, 37) can appear in the first column on our example page, we can specify a given value in this range by using only 3 bits: 000 represents 33, 001 represents 34, 010 represents 35, and 011 represents 36 and 100 represents 37.

Consider a set S of points or rectangles and collect, from S, the minima and maxima for all dimensions in S. The minima and maxima provide a **frame of reference** F, in which all the points/rectangles lie. For instance, if

$$S = \{(511, 1001), (517, 1007), (514, 1031)\}$$

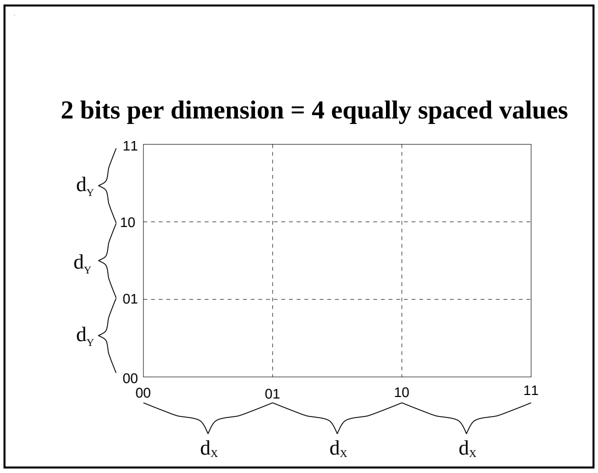
then

$$F = [511, 1001] \times [517, 1031]$$

The frame of reference tells us the range of possible values in each dimension for the records in the set S. For instance, the points along the X-axis only vary between 7 values (511 to 517 inclusive), and the points along the Y-axis vary between 31 values. Only 3 bits are actually needed to distinguish between all the X values that actually occur inside our frame of reference, and only 5 bits are needed to distinguish between Y values. The set of points S would be represented using the following bit strings:

$$S = \{(000, 00000), (110, 00110), (011, 11110)\}$$

Since the number of records stored on a page is typically in the hundreds, the overhead of remembering the frame of reference is well worth it: in our example, if values were originally stored as 32 bit integers, we can compress these points with no loss of information by (on average) a factor of 4 (without taking into account the overhead of storing the frame of reference)!



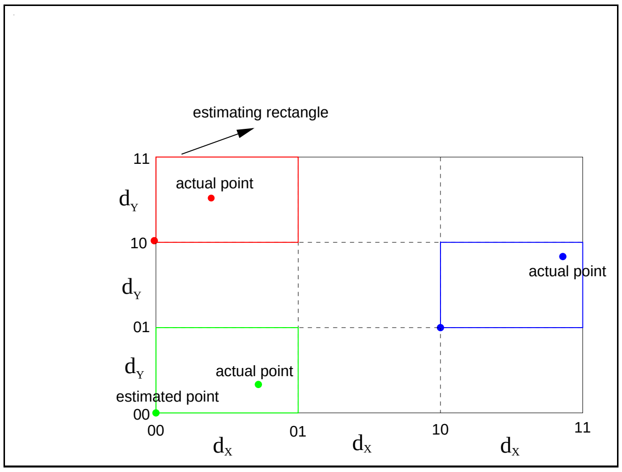


Figure 1: Frame of reference for lossy compression. Figure 2: Point approximation in lossy compression.

Sometimes, it is sufficient to represent a point or rectangle by a bounding rectangle, e.g., in the index levels of an R-tree. In this case, we can reduce the number of bits required as much as we want by trading off precision. The idea is that we can use bits to represent equally spaced numbers within the frame of reference (see Figure 1), thereby creating a uniform 'grid' whose coarseness depends on the number of bits used to represent 'cuts' along each dimension. Each original point or rectangle is represented by the smallest rectangle in the 'grid' that contains it. If the original data consists of points, these new rectangles are always of width 1 along any dimension, and we can represent such a rectangle by simply using its 'min' value along every dimension, e.g., the lower left corner in two dimensions (see Figure 2). For instance, if 2 bits per dimension were used for both the X and Y axes on S,

$$S = \{(00, 00), (11, 00), (01, 11)\}$$

2.2 File-Level Compression: Bulk-Loading

We present three bulk-loading algorithms in this section. Each of these algorithms produces an index structure (R-tree or B-tree) containing the data records in the leaf pages. The first algorithm, called GBPack, is based on the STR [LEL96] bulk-loading algorithm for R-trees. The second algorithm is a variant of the first and trades off some compression to obtain a higher-quality R-tree index. The third algorithm is also a variant of the first, and produces a B-tree index whose search key is the concatenation of all the attributes used for sorting during the bulk-loading. ([NR95] discusses the use of such B-tree orderings.) We describe the first bulk-loading algorithm in detail and then describe the other two briefly as variants of the first.

GBPack: Compression-Oriented R-Tree Bulk-Loading Algorithm

All bulk loading algorithms in this paper partition a set of points (center points, if the data contains rectangles) into pages. The partitioning problem can be described as follows:

Input. A set (or multiset) of points in some n-dimensional space. We assume that each dimension (axis) of that space has a linear ordering of values.

Output. A partition of the input into subsets. The subsets are usually identified with index nodes or disk pages.

Requirements. The partition should group points that are close to each other in the same group as much as possible. The partition should also be as unbiased as possible w.r.t. (a set of specified) dimensions.

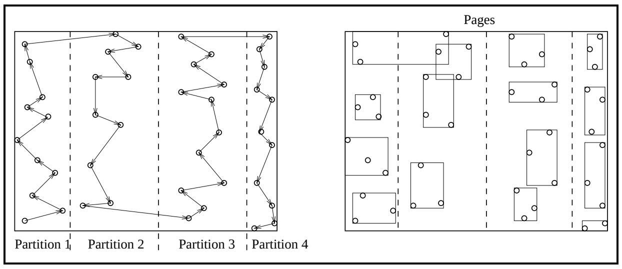


Figure 3: Example for the bulk loading sort operator (Using GBPack).

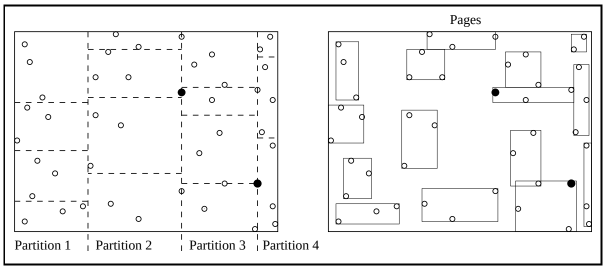


Figure 4: Example for the bulk loading sort operator (Using STR). Bold points belong to partitions to the right of the point.

The problem above can then be reapplied to the resulting hyper-rectangles corresponding to the pages produced by the above process. This reapplication is repeated until all the references to the resulting pages along with their hyper-rectangular bounds can be fit on one page, the root.

The following example in two dimensions demonstrates the GBPack algorithm. Consider the set of 44 points shown as small circles in Figure 3. Suppose we determine that the total number of pages needed is 15. We sort the set on the X dimension in ascending order. Then we define $p := \lceil \sqrt[2]{15} \rceil = 4$ as the number of partitions along the X dimension; taking the square root reflects our assumption that the number of partitions along each of the two dimensions is equal (i.e., we expect each of the X-partitions to be cut into 4 partitions along the Y dimension). We sort the first partition on the Y dimension in ascending order. Then the second partition is sorted in descending order, the third in ascending order and the fourth in descending order. Figure 3 shows the partitions. The arrow in the figure shows the general ordering of points for that dataset. Note that the linearization generated by the alternation of sort order guarantees that all but the last page are fully packed at the expense of a little spatial overlap amongst the leaf pages. In the above description, we assumed that the number of pages needed was known to be 15. This number was then used to determine the number of partitions along each axis. In actuality, we don't know the final number of pages needed since it depends on the compression obtained, which depends on the data. Therefore, we use an estimate of the number of pages, obtained by assuming that we have the bounding box of all the data and values in tuples are uniformly distributed over this range.

Finally, in the case of low cardinality data, we want to guarantee that the partition divisions happen along changes in value of the dimension being cut. This results in a much better division of the partitions into small ranges when one considers the degenerate case of low cardinality data. This is a result of narrowing the ranges over all the pages in the dataset, since the same value isn't in more than one partition (since the partitions don't contain overlap). For instance, in figure 4, note that one of the natural partition divisions occurred between two points that had the same X value. Nonetheless, we did not make the partition there since it would have reduced compression of our dataset. See Section 2.3 for more details.

Our partitioning technique is similar to STR in that, starting with the first dimension, we divide the data in the leaf pages into 'strips'. For instance, figure 4 shows how STR decomposes the data space into pages. Since, in STR, we are sorting uncompressed data, we can calculate exactly the number of pages P produced by the bulk loading algorithm. P, in this case 15, is used to calculate the number of pages in each strip (except the last). In this case, we determine that the first three strips have four pages, while the last has three. Thus, since the first three strips contain four pages each, each strip contains 4*3=12 points each. The last strip contains whatever points are left (8 in this case). Each of the strips are then grouped into partitions with 3 entries each, except the very last page, which contains 1 point. Note that all pages are fully packed except the last.

The important differences between our algorithm and STR arise from three considerations:

- We are using our bulk loading algorithm to pack data onto compressed pages, and are willing to trade off some tree quality for increased compression.
- The degree of compression is based on the data on a given page, and this makes the number of entries per page data-dependent.
- In the case of low cardinality data, it is very important that when we cut a dimension, it is done on a value boundary in the data.

The first item listed above simply means that we pack pages more aggressively at the expense of increased spatial overlap among the partitions. We do this by creating a linearization of the data that allows us to 'steal' data from a neighboring partition to fill a partially empty partition. In particular, if one considers the above example, there is a partially empty page at the top of each strip. These pages can be filled when one considers the effect of reversing the sort order of each strip. The results are illustrated by Figure 3. Once this linear ordering is achieved, the data may be packed onto pages from the beginning of the linearization to the end. The second point means that we have to estimate the required number of pages. The third point constrains how we determine partition boundaries.

We now present the GBPack algorithm in more detail. The ordering of points is determined by a sorting function $sort(A, k, D)$. The arguments are:

A: An array of items to be ordered. This is a 2 dimensional array of integers where the first array index is tuple number and the second identifies a particular dimension (i.e. $A[i]$ is tuple i, $A[i][j]$ is the value of the jth dimension for the ith tuple). we use the notation $|A|$ to refer to the number of tuples in A.

k: The number of elements in A.

D: An integer identifying the dimension we want to sort on.

In addition, there exists the following global variable:

S: An array of d booleans where d is the dimensionality of the data.

If $S[d]$ is true, the current sort order for dimension d is ascending, otherwise it is descending. The initial value of S does not matter.

The function $sort(A, k, D)$ has the following steps:

1. $S[D]$ = not $S[D]$. Reverse the sort order for every dimension. This step ensures the linearization of the space illustrated in the example later in this section.

2. Sort the array A on dimension D according to the sorting order $S[D]$.

3. If $D \neq 1$ do

   (a) Estimate P as the number of pages needed for storing the items in the array. This estimate is more difficult since the frame of reference for an individual page is needed to determine the compression ratio. Currently we estimate this number by retrieving all tuples to partition and using their frame of reference and the number of tuples to get a very accurate estimate of the number of pages.

   (b) Set $p := \lfloor \sqrt[D]{P} \rfloor$. This is the number of partitions on dimension D for the array.

   (c) Define a list L s.t. for $0 \leq j \leq d$ $L_j = \frac{j \cdot k}{p}$. L is the list of partition start locations.

   (d) Lower all values in L s.t. a particular L_j is the first occurrence of $A[L_j][D]$ in A.

   (e) Remove any duplicates from the L array. This and the previous step guarantee that dimensions aren't overcut; an important guarantee for low cardinality fields.

   (f) For $1 \leq j \leq |L| - 1$ do the following:
   $sort$(array starting at $A[L_j], L_{j+1} - L_j, D - 1)$.

   (g) $sort$(array starting at $A[L[|L|]], k - L_{(j)}, D - 1)$.

To perform bulk-loading of an entire tree to maximize compression, we use the function $sort(\mathcal{A}, |A|, D)$. Since the resulting array A is now sorted, we simply pack the pages of the leaf level maximally by introducing entries sequentially from the array until all entries are packed. This process is repeated for higher levels in the tree by using the center points of the resulting pages in the leaf level as input to the next level. Observe that both leaf and internal nodes are compressed in the resulting R-tree index.

2.3 GBPack: Quality-Oriented R-Tree Bulk-Loading Algorithm

To perform bulk-loading of an entire tree to maximize R-tree quality, the deepest level of the recursion (1D strips) writes out pages instead of sorting the entire array first. This, when combined with partitioning only along changes of value, results in lower page occupancy, and guarantees no overlap amongst leaf pages over point data. Note that it still handles low cardinality attributes more gracefully than STR.

2.4 B-Tree Bulk-Loading Algorithm

To compress the data records by ordering them as in a B-tree, the GBPack algorithm can be used to sort along all dimensions at once (all dimensions are taken together and treated as one, concatenated, search key). What results is a compressed, clustered, primary B-tree over the concatenated search key. The algorithm in this case is actually much simpler than the general version presented above, and runs much faster, but we will not elaborate upon this for lack of space.

3 Compression Applied to Rectangle Based Indexes

Many indexing structures, including R-trees [Gut84], B-trees [BM72] , grid files [NHS88], buddy trees [KSSS90, KSSS89, KHS91], TV-trees [LJF94] (using L_1 metric), and X-trees [BKK96], all consist of collections of *(rectangle,pointer)* pairs and *(point,data)* pairs. Our compression technique can be used very effectively on these indexing structures, and is especially useful when the search key contains many dimensions.

While the behavior of our compression technique when used in index structures is similar in some ways to B-tree prefix compression, our compression scheme is different in that:

- We translate the minimum value of our frame of reference to 0 before compressing.
- Lossy compression makes better use of bits for internal nodes than prefix compression since all bit combinations fall inside our frame of reference.
- We also compress leaf level entries, unlike prefix compression, which is applied only at non-leaf nodes.

The following sections discuss the impact of compression on R-trees. This includes the alterations needed for dynamic insert, delete, and update. In addition, internal nodes are compressed using lossy compression.

3.1 Description of R-trees

The R-tree [Gut84] is a height-balanced tree structure designed specifically for indexing multi-dimensional spatial objects. As in a B-tree, each node (a disk page) in the R-tree is either an internal node or a leaf node. Entries in internal nodes point to other nodes. Entries in leaf nodes contain the object IDs of the actual spatial objects (or, optionally, the objects themselves), as well as the *minimum bounding box* (MBB) of the object. Each internal node entry also contains an associated MBB; this is guaranteed to completely cover all MBBs in the subtree pointed to by the entry. Note that two entries in the same node might have associated MBBs that overlap. This means that unlike B-tree searches, an R-tree search may explore more than one path from the root to a leaf, even for point queries.

All MBBs are oriented orthogonally with respect to the (fixed) axes for the R-tree. If the number of dimensions is greater than two, MBBs with an appropriate number of dimensions are used, but the basic idea remains the same.

The above description is sufficiently general that it applies to several variants of the R-tree, including R^+-trees [SRF87] and R^*-trees [BKSS92]. (Note that MBBs in non-leaf nodes of an R^+-tree are non-overlapping, and therefore point queries search a single path from root to leaf. Nonetheless, region queries must explore several paths, in general.) These variations were motivated by the goals of improving space utilization, and therefore also search time (by reducing tree height and minimizing MBB overlaps); we will not discuss the variations further.

3.2 Compression Within R-tree Internal Nodes

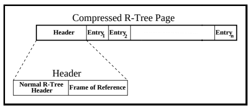


Figure 5: Compressed R-tree page layout

Since the internal nodes of R-trees simply store *(hyper-rectangle, page pointer)* pairs, compression is used to compress the hyper-rectangle stored in each pair. As a result, the header for the internal node must now include the frame of reference for all hyper-rectangles on the page (see Figure 5). Potentially lossy compression using a global number of bits per dimension can be used in all internal nodes to ensure fixed size entries in internal nodes. (This simplifies the implementation of the insert algorithm, in particular splitting.)

Given this representation for an internal node, modifications to the R-tree can cause a change in the frame of reference for an internal node. For instance, suppose a point is added to a region not represented by an R-tree. A path from the root to a leaf must be created that covers the space which the point resides in. This involves widening the range of values for pages along that path. When one of these changes of reference occur (actually, this only happens in insert and modify, not delete), we must convert all entries from the old frame of reference to the new one. This can be done with the following:

BitsPerDim : The bits per dimension for each dimension of entries on the page

Boxes : an array of uncompressed hyper-rectangles

Entry(i) : the i-th entry on our node

Compress(Box, Frame, BitsPerDim) : returns the compressed representation of a box

Uncompress(Box, Frame, BitsPerDim) : inverse of Compress

```
InternalNode::ChangeFOR(NewFrame)
  for every valid entry number EntryNum in our node
    Boxes[EntryNum] = Uncompress(Entry[EntryNum].box_part, CurrentFrame, BitsPerDim)
  CurrentFrame = NewFrame
  for every valid box number BoxNum in Boxes
    PAGE.entry[BoxNum].BoxPart = Compress(Boxes[BoxNum], CurrentFrame, BitsPerDim)
```

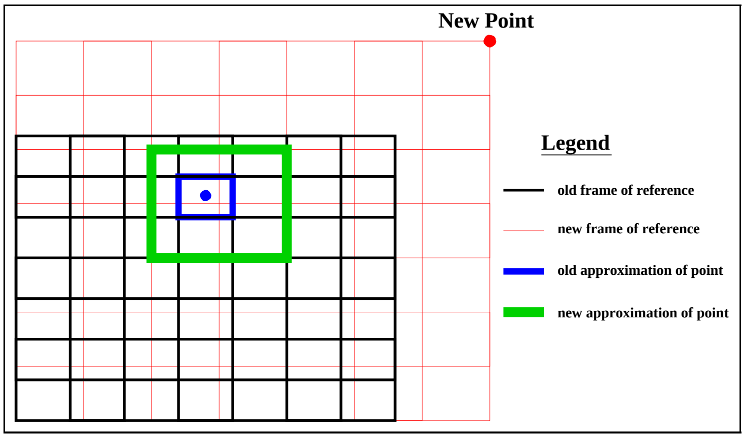


Figure 6: Change of approximation of point after frame of reference changes

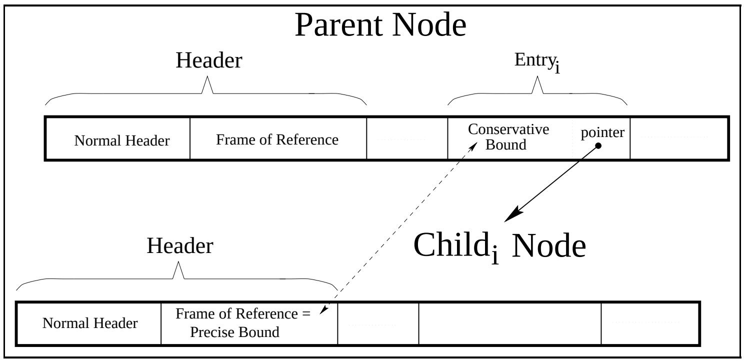


Figure 7: Conservative bounding box in parent is tied to exact bounding box in child

Note, from the algorithm above, that changes of references are done by widening the rectangles (see Figure 6). This can lead to very large successive approximation error. To alleviate this problem, several steps must be taken. These steps use the frames of references in the children as bounding boxes for the actual data in the child. The corrective steps taken are:

1. When following child pointers down the tree during insert, the frame of reference for a child is always used to update the corresponding entry in the parent node (see Figure 7).

2. The number of times the frame of reference has been changed is stored in the header. This information is used to determine when occasional cleanups occur, where a cleanup consists of getting minimally lossy bounds for the entries. These minimally lossy bounds are determined by examining the frames of reference for all children of the page being cleaned.

Note that the cost of Step 1 is virtually nil since no additional I/O is done. In practice, this step has an enormous effect on reducing successive approximation error. Step 2 is much more expensive since it involves loading all the children of the node being cleaned, and, therefore, should be done much less frequently. A discussion of how frequently cleanup must be run is deferred until later.

3.3 Compression Within the R-tree Leaf Nodes

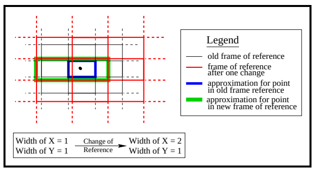


Figure 8: The change in approximation after one change to frame of reference

Note that if the index is a secondary index, the global bits per dimension scheme mentioned for use in internal nodes can apply to leaves as well. In this situation, the only differences between leaf nodes and internal nodes arise from successive approximation error, which cannot be fixed using corrective step 1 above. Thus a slightly modified version of step 2, which retrieves the actual data from the database, is the only available method to correct error. Again, a discussion of how frequently cleanup must be run is deferred until later.

If the index is a primary index, the bits per dimension for each page are individually determined by the data on that page. No cleanup is ever necessary since no information is lost.

Leaf nodes of secondary indexes over point data: Note that if the index is a secondary index, and the actual data is points, changes in the frame of reference result in widening of the box implicitly used to approximate the point (see Figure 8). At first glance, this seems to indicate that a box representation of the point must be used, resulting in a doubling of entry size! Upon further reflection, this is unnecessary. The width of the box used to represent a point is exactly one more than the number of frame of reference changes since the last cleanup. This is easy to see when considering the scenario depicted in Figure 8.

4 Performance Evaluation

This section tests the effectiveness of our compression technique in a variety of situations. Section 4.1 focuses on the effectiveness of our compression strategy when used to compress relations. We explore the compression over both real and synthetic datasets. In addition, the appropriateness of our compression strategy is examined for B-trees. Section 4.5 examines the performance gains when our compression technique is applied to R-trees. Section 4.1.4 demonstrates our compression techniques applied to the Tiger GIS dataset. Section 4.2 discusses the CPU costs associated with decompression in comparison to gzip. Section 4.3 compares our compression techniques with those found in Sybase IQ. Finally, Section 4.4 gives some examples in which cheap tuple level decompression significantly enhances performance.

4.1 Relational Compression Experiments

Since relational file compression improves both space utilization and linear scan performance, the experiments here measure the size of the compressed file as compared to the original in a variety of situations. Compression is examined under several tuple partitionings, including maximum occupancy random partitioning, maximum compression R-tree (STR) partitioning (using the modified STR algorithm), maximum quality R-tree (STR) partitioning, and B-tree partitioning.

All results are shown in compressed file size as a percentage of the original file size. For example, when we get a 25% compression ratio it means that the space taken by the relation was reduced by a factor of 4.

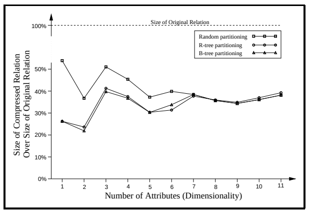


Figure 9: Sales dataset. Compression achieved by varying dimensionality and partitioning strategy.

Since, we want to examine the effect of partitioning strategy on relation compression, we conducted an experiment in which we studied the partitioning strategy over a real dataset. For three partitioning strategies, namely random, R-tree optimized for compression and B-tree, we varied the dimensionality of the datasets while keeping the number of tuples constant. The results are shown in Figure 9. Note that B-tree partitioning and R-tree partitioning, while producing almost identical compression results, are significantly better than random partitioning. This demonstrates the value of spatial grouping as the basis for tuple distribution over pages.

In addition, note that as dimensionality increases, the effect of smart partitioning is reduced. The underlying reasons for this trend become clear when one thinks about the effect of grouping close points together: In order to reduce our key size by some percentage, we must reduce the size of each dimension by the same percentage. Suppose that percentage determines that we need a reduction of 1 bit in each dimension. We must, therefore, divide the range of every page along all dimensions by a factor of 2. This means we must divide the data into 2^n pages to obtain our improvement due to smart partitioning! Thus the benefit of smart partitioning decreases exponentially w.r.t. dimensionality. As a result, as dimensionality was increased, the benefit of smart partitioning was reduced to nil for the higher dimensions.

4.1.1 Synthetic Data Sets

In this section, synthetic datasets were used to test our compression scheme under a variety of conditions. The following is a list of variables in our experiments:

Size: The number of tuples in the relation.

Dimensionality: The number of attributes of the relations.

Range: The range of values for the attributes.

Distribution: The distribution of values for each attribute was varied using both uniform(worst case) and exponential distributions.

Partitioning strategy As described in Section 2.2 we have three strategies. These strategies are: maximal compression for R-tree partitioning, maximal quality for R-tree partitioning, and B-tree partitioning.

Page size. Since we varied page size from 1KB to 8KB and found only slight differences, all presented experiments were performed on a page size of 4KB. Increasing page size beyond 4KB had a slightly negative effect on compression ratio.

Each dataset had the following characteristics:

- All attributes for a particular experiment had identical distributions (created by the same randomizing process).
- Attributes were statistically independent.

The figures in this section occur in pairs, where the left figure refers to an experiment using a uniform attribute distribution and the right figure refers to the same experiment using an exponential distribution.

Figures 10, 11 and 12 are collected together and shown at the end of the paper for easy comparison— shows the compression achieved on a file size of 100000 tuples and uniform attribute distributions with high compression R-tree partitioning. We varied both dimensionality and range. The different result lines represent datasets with varying attribute range. We assumed that all attribute values were represented in the original relation using 4 bytes. Since, as the range of the attributes decrease, the frames of reference for individual pages narrows, compression improves as attribute range decreases. More specifically, if we examine the top line in the graph, the range of values covers 24 bits of the 32 bit integer in the original dataset. As a result, we expect to compress the file to at most 75% of its original size. Note that compression is significantly better for the exponentially distributed data. This is due to a large part of the data falling into a narrower range, resulting in smaller frames of reference on average.

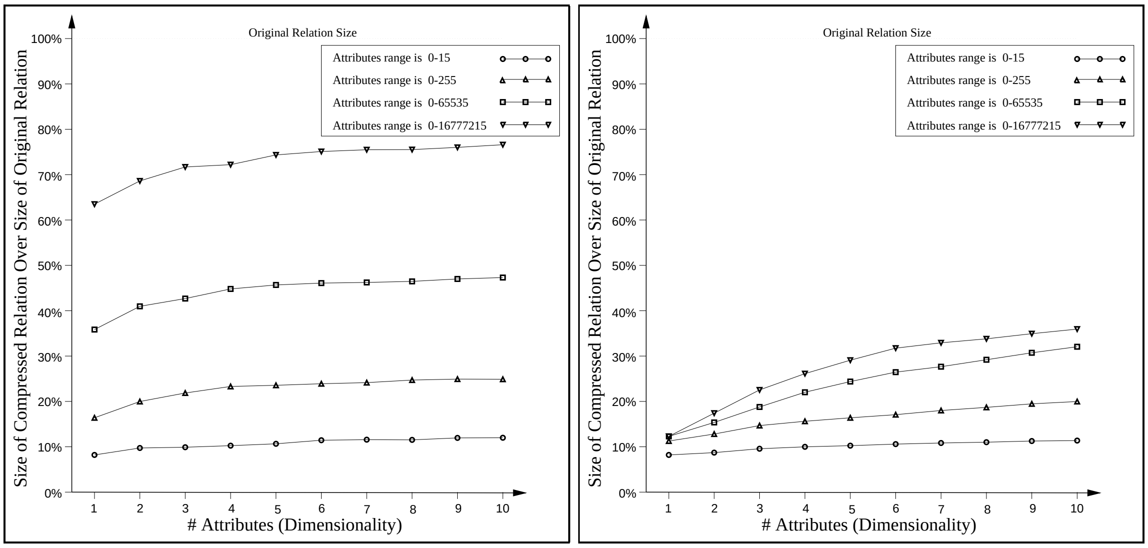


Figure 10: Results for R-tree bulk loading geared for compression.

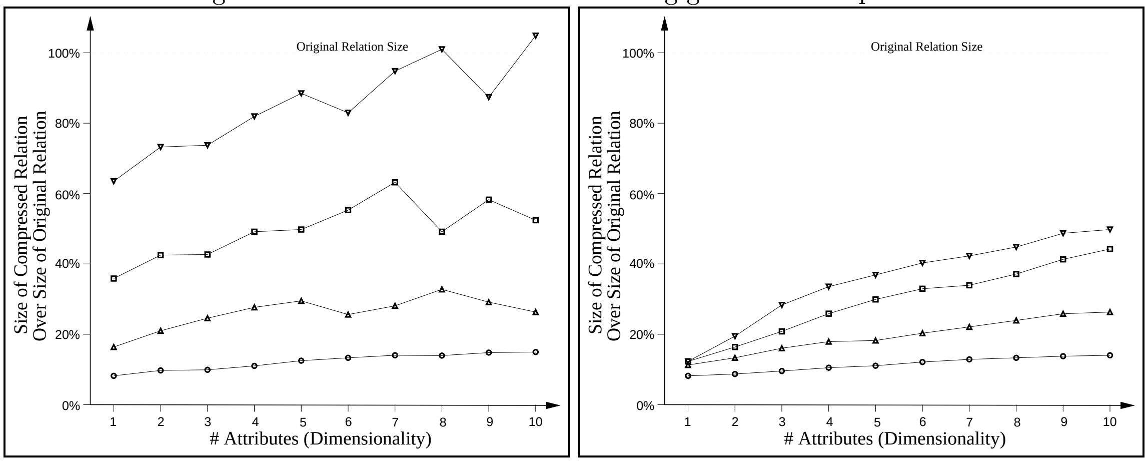


Figure 11: Results for R-tree bulk loading geared for R-tree quality.

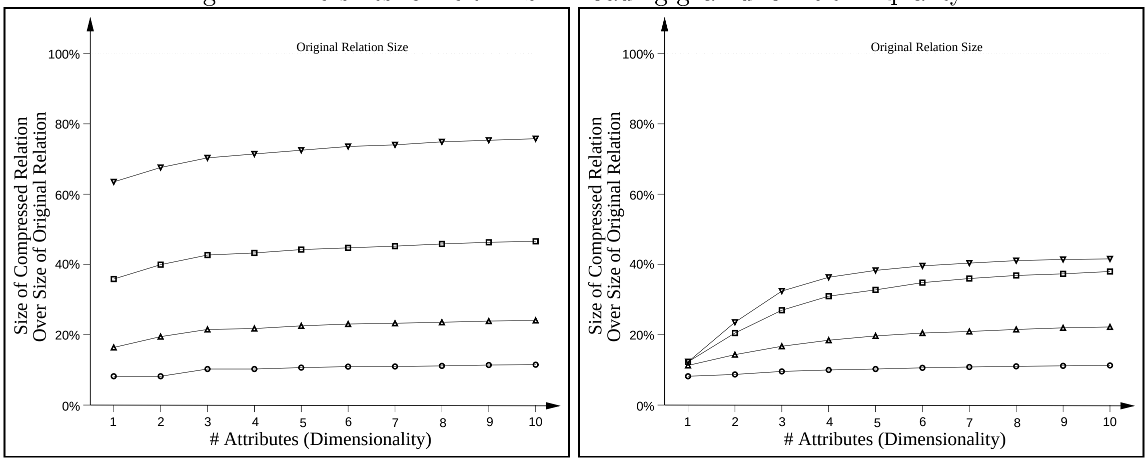


Figure 12: Results for B-tree bulk loading.

Figures 11 and 12 show results for the exact same datasets as Figure 10. The difference is that we use the partitioning for R-tree quality and B-tree ordering respectively. Note that the B-tree partitioning and R-tree high compression partitioning are nearly identical while the R-tree quality partitioning lags slightly in the higher dimensions. Note that in all the experiments with skewed (exponential) distributions, the compression worked remarkably well.

Figure 13 shows results for the same experiment as Figure 10 while using 500000 tuples instead of 100000. Note the slightly improved compression. This is understandable when one considers that the number of cuts made in STR is based upon the size of the input, and in this case, results in about 2 bits of reduction in total key size.

The results in the previous experiment motivated another experiment whose results are shown in Figure 14. In this experiment the number of tuples in a 2 dimensional dataset was repeatedly doubled. Note the significant improvement in compression with each doubling.

4.1.2 Companies Data Set

This section describes compression experiments on a sample of the Compustat data set. This dataset's attributes describe many aspects of companies on a stock exchange. Some of these attributes include asset value and share price. The sample contained 82600 tuples and about 300 attributes. In this experiment all indexes were partitioned for high R-tree compression. We projected 10 attributes from the relation. Many of the fields in this dataset, while having very high range, were also extremely skewed.

Figure 15 shows the compression results when we varied dimensionality and page size. (We varied dimensionality by using a further projection on the 10 attribute dataset.) All tuples were used in all experiments. We achieved compressed file sizes between 18% and 30%. Note that, as asserted in the previous section, page size had little effect on compression.

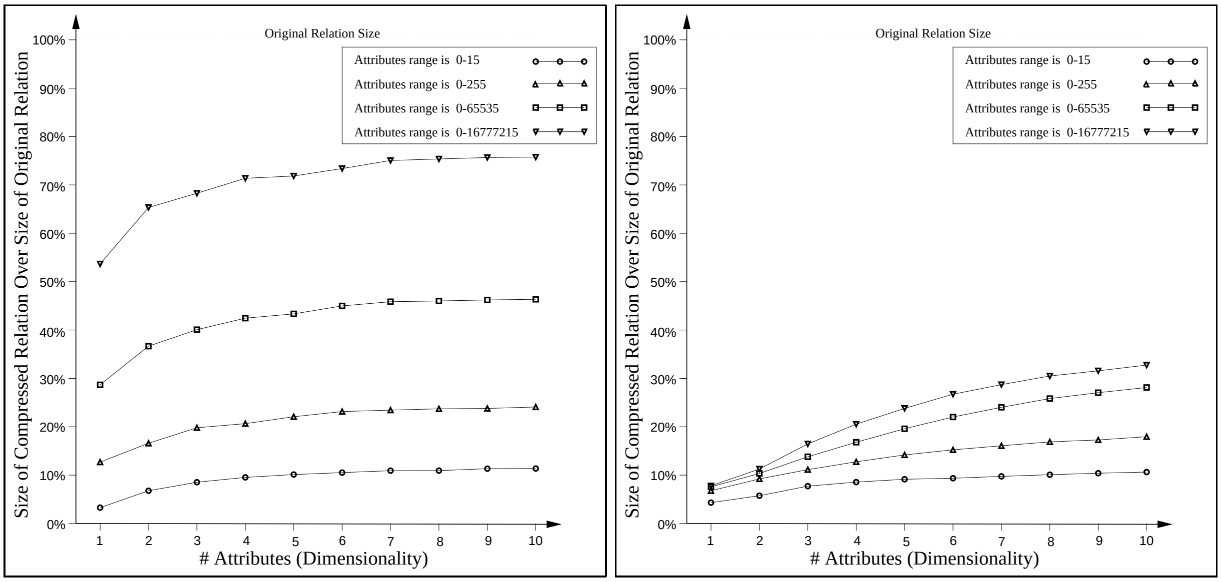


Figure 13: Synthetic dataset, 500000 tuples, R-tree bulk loading geared for compression. Left graph is for uniformly distributed data. Right graph is for exponentially distributed data.

4.1.3 Sales Data Set

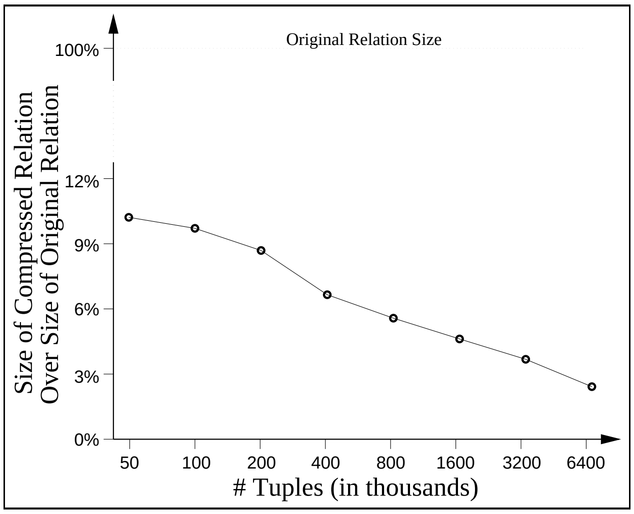


Figure 14: Synthetic dataset, changing number of tuples.

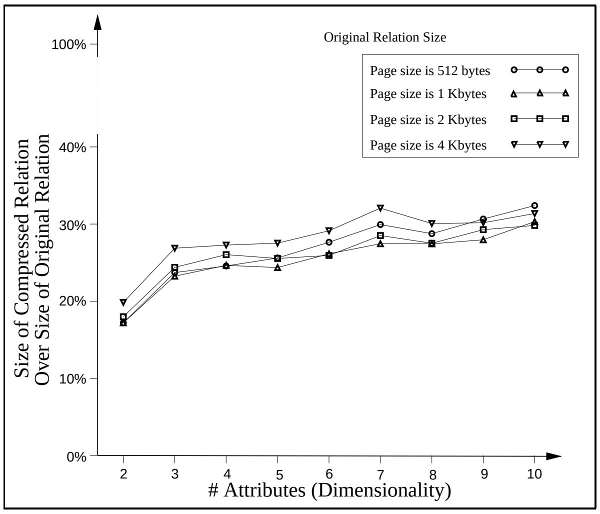


Figure 15: Relational compression for Compustat dataset.

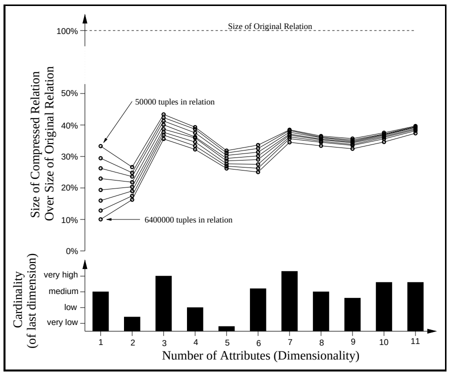


Figure 16: Sales dataset. Compression achieved versus dimensionality.

This section describes relational compression experiments done on a sales dataset. The dataset is taken from a catalog sales company and has eleven attributes. Four attributes were low cardinality. For each of those attributes we created a table that maps the values possible for that attribute to the range $0, .., k$ where $k+1$ is the number of possible values. Then, we use the mapped values in the relation instead of the original values.

All experiments on the sales dataset were partitioned for R-tree high compression. The experiment in this section, whose results are shown in Figure 16, examined compression as dimensionality was increased by taking progressively larger projections of the original dataset. The cardinality of the added dimension is shown for each dimension. Note that when a low cardinality attribute is introduced, the compression improves while a high cardinality attribute reduces compressibility. This is easily understood since low cardinality attributes need fewer bits to represent. In addition, the different data lines represent different dataset sizes. Note that the effect of file size is diminished with dimensionality.

The compression we see in Figure 16 is quite good. In comparison, gzip, which uses Lempel-Ziv compression, when applied to the file in BTree sort order, was able to compress the file slightly better(about 6Of course we maintain the notion of a tuple ID and guarantee light decoding costs for single tuples while gzip does not.

4.1.4 A GIS Data Set

In this experiment, we used the Tiger dataset for Orange county, California. The objects in the data are polygons (roads, rivers, etc.). We used 150000 tuples. When partitioning for R-tree compression we got 54% compression (i.e., the resulting relation size is 54% of the size of the original data file). When partitioning for R-tree quality we got 55% compression.

When the file was unsorted, we got only 68% compression. By comparison, gzip compressed to 96% of its original size when the dataset was unsorted and to 61% of its original size when sorted. Therefore, for this dataset, we achieved better compression in all situations.

4.2 CPU vs. I/O Costs

In this section, we explore the CPU costs associated with decompressing information on compressed pages. For comparison's sake, we also present CPU costs associated with gzip when used in a manner consistent with Sybase IQ.

All experiments conducted using our compression techniques extracted information from the first hundred pages of both the CompuStat dataset. In both cases, all 100 pages were in memory compressed, resulting in no I/O costs. Our measurements included the time to scan the contents of a full compressed page (Everything), the time to scan one field of all tuples on a full compressed page (Field), the time to scan one tuple on a full compressed page (Tuple), and the time to count the number of entries on each page (Count). The corresponding throughputs are also given (i.e. the rate at which a disk needs to supply information to the CPU to keep the CPU busy). Note that all of the code that was used for these experiments used an endian/platform independent bit string package. These numbers could be improved significantly by writing the code in platform specific assembly language. The results are in Table 1

	Everything	Field	Tuple	Count
Processing Time	0.256ms	0.099ms	0.013ms	0.047ms
Throughput	16MB/s	41MB/s	305MB/s	88MB/s

Table 1: Our Compression Results

All gzip experiments were performed by dividing the uncompressed dataset into 64KByte blocks, which were then compressed. Note that the resulting compressed blocks ranged in size between 4KBytes and 32KBytes. The CPU time was then measured for decompressing all blocks within each size category (4K, 8K, 16K, 32K). To ensure that any overhead in starting gzip was taken into account, the time to decompress the same number of compressed 1Byte blocks was subtracted from the measured times. Note that since field and tuple level decompression isn't possible with gzip style compression, only the cost associated with decompressing an individual block is measured. The corresponding throughputs are also given (i.e., the rate at which a disk needs to supply information to the CPU to keep the CPU busy). The results are in Table 2.

	4KBytes	8KBytes	16KBytes	24KBytes	32KBytes
Processing Time	5.6ms	6.6ms	9.6ms	11.2ms	14ms
Throughput	0.7MB/s	1.2MB/s	1.7MB/s	2.1 MB/s	2.3 MB/s

Table 2: Gzip Compression Results

Note from the above tables that our decompression is a great deal faster than gzip. While we can easily keep up with a linear scan in all situations (approximately 5MB/s), gzip doesn't keep up under any of the measured circumstances. In addition, there is a large performance benefit in accessing individual tuples or fields of a page, while gzip is unable to take advantage of such situations. See Section 4.3 for more information about the advantages/drawbacks of gzip style compression in comparison to ours.

4.3 Comparison With Techniques Found in Commercial Systems

Many commercial vendors are utilizing compression techniques in their data warehousing products. We discuss Sybase IQ in particular because their techniques are among the best in current commercial systems, and published information on the use of compression in commercial DBMS products is generally lacking. Sybase compresses data using a technique similar to gzip; in the rest of this section, we will simply say 'gzip' for brevity. Note that if all values of all fields are listed in the same order w.r.t. tuple ID, tuple IDs don't need to be stored with the projection. Sybase also uses bit vectors, and applies gzip to compress each bit vector.

Note that the use of gzip has advantages and drawbacks. One advantage is that gzip is a general compression technique and can be used to compress any type of field, including strings. The drawback of gzip is that it does not support random access to tuples within a page, and decompression is relatively slow. This means that pages being actively used must be stored in memory uncompressed. Another drawback is that applying gzip page-at-a-time causes some complications, which we outline below. (Applying gzip file-at-a-time would mean that if we need a particular tuple, or page, we would have to decompress the entire file!) (An important exception to this problem occurs during linear scans. During linear scans, processed pages can be "tossed" immediately.)

A consequence of applying gzip page-at-a-time is that since different pages compress to different extents, the result of compressing a page is of variable size. Since pages in the buffer pool are of some fixed size, we need to map variable sized "compressed pages" to fixed size "logical pages". In Sybase IQ, pages of 64K are compressed by applying gzip, and each compressed page is rounded up to a 4K, 8K, 16K, 32K or 64K "logical page". (When a logical page is uncompressed, it yields a 64K page of the original file.) This means that there is empty space at the end of a logical page, which affects the compression that is obtained. Logical pages are allocated sequentially on disk, and prefetched in 64K increments. If random access to a page in the original file is desired, a mapping from logical pages to disk pages must be maintained.

In summary, gzip provides high flexibility in the type of data compressed while complicating buffer and storage management. These complications lead to inefficiencies in memory and disk utilization. Another drawback of gzip is that gunzip can only sustain an uncompression throughput of 1.7MB/s (see Section 4.2) on a 200MHz Pentium Pro. This uncompression rate is not high enough to keep up with a linear scan. Note that this may be improved with a more efficient implementation of gunzip—Sybase IQ uses a proprietary version of gzip from Stacker, not the standard Unix gzip—but we do not have access to an optimized gzip implementation.

We note that Sybase IQ stores relations column-wise. While this is likely to improve the compression obtained using gzip (or other compression techniques, including ours), the decision of whether to vertically partition a relation is in general orthogonal to whether to use compression. If vertical partitioning is used, and columns are stored in the same order as the original set of tuples, our compression technique can be used to compress columns without storing tuple IDs with each field value. On the other hand, if a column is reordered (e.g., using B-tree ordering), tuple IDs must be stored with each field value in the column; this is the case in either the Sybase approach or ours, and in either case the tuple ID is compressed too.

The table 3 summarizes the resulting size of compressed files when using Sybase IQ vertical partitioning with compression; compressed bitmaps; and our compression technique. In all cases, the values of each tuple in each field were mapped to a number between 0 and n-1 where n is the cardinality of that field. For the Sybase experiments, gzip is used with the uncompressed buffer page size 64Kbytes and the compressed size either 4Kbytes or a multiple of 8KBytes. The page size used in our experiments was 4Kbytes. These experiments were done with columns stored in the same order as the original tuples, so tuple IDs weren't stored with each value.

	Sybase Vertical	Sybase Bit vectors	Ours
Low Cardinality Sales	10%	7%	1%
Medium Cardinality Sales	23%	NA[2]	37%
Compustat	29%	NA[2]	30%

Table 3: Sybase IQ compression comparison

Note that Sybase's techniques typically result in comparable performance. Of note is the low cardinality sales data, where the 1/16 compression limit that results from the "impedance mismatch" of buffer pool and disk pages limits compression.

4.4 Importance of Tuple-Level Decompression

In this section we present two scenarios where fast tuple level decompression significantly enhances performance. The first example is an index nested loops join where the inner relation's index (a compressed B-tree) and the compressed inner relation fit in memory, but the uncompressed inner relation does not fit.

In this scenario, the tuples from the outer relation are brought into memory one page at a time. Each tuple from the outer is then used to probe the inner through the index, and the cost of the join is dominated by the cost of retrieving matching inner tuples. If either the inner or the outer isn't sorted on the join column, each probe matches tuples that are randomly distributed across the inner relation; thus, we need just a few tuples from each page containing a matching tuple. In our approach, the rid of a matching tuple identifies the (compressed) inner page and slot containing the tuple, and locating and selectively decompressing the tuple is extremely fast. If page-at-a-time gzip compression is used, the rid of a tuple in the uncompressed inner relation must be mapped to the logical page containing the tuple (or the compressed page containing the tuple must be identified somehow), and this logical page must be decompressed in its entirety. The difference in CPU costs between this and our approach to retrieving a matching tuple is a factor of about 500 (see Section 4.2). Since the entire compressed inner is assumed to fit in memory, there are no I/O costs, and this difference dominates the relative costs of computing the join by using our compression or gzip to compress the inner.

Alternatively, we can keep the uncompressed inner relation on disk, and bring in pages containing matching tuples for each probe, or use other conventional join techniques, but the algorithm outlined above will be superior because the total I/O is just one scan of the outer plus the (compressed) inner, plus the CPU cost of individually decompressing each matching tuple; we've seen that the CPU cost is very low.

A second scenario in which tuple level decompression is beneficial occurs in a multiuser system where the dataset fits in memory compressed, but not uncompressed. In this scenario, it is easy to imagine that many people will be simultaneously accessing random tuples within the database. For instance many people could be performing exact match queries, or queries which use indices to retrieve small portions of the database. Note that even substantial queries (e.g. 5selectivity) using secondary indices result in random access to the underlying pages in the relation. (Similar scenarios can be constructed for a single-

[2]This experiment wasn't performed since there were several medium cardinality attributes which make the use of bit vectors inappropriate.

user environment as well, by using a different workload.) Being able to store pages compressed and extract individual tuples rapidly is the key to good performance here.

4.5 R-tree Compression Experiments

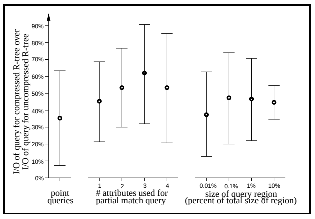


Figure 17: Testing the quality of R-trees on Sales dataset.

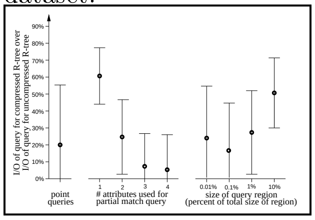


Figure 18: Testing the quality of R-trees on Compustat dataset.

This section examines the overall impact of compressing R-tree pages on R-tree performance. Note that compressing the leaf pages of indexes results in size improvements comparable to general relational compression. The internal nodes benefited from significantly improved fanout.

Since the experiments in Section 4.2 determined that compressed pages can be decompressed as fast as current disk drives doing sequential access, queries over R-trees, which exhibit random access characteristics, are bounded by the I/O costs associated with getting pages from disk. As a result, only the page I/O is measured for these experiments.

These experiments used the three real datasets already discussed to test the quality of a compressed R-tree (using R-tree quality partitioning) vs. the quality of an uncompressed R-tree. For that purpose we performed identical queries over both compressed and uncompressed R-trees, and measured the page I/O caused by each query. For the purposes of making our results more realistic, we assumed that the root of each tree remains in memory and doesn't incur I/O costs. Each query type was run 100 times. Using the page I/O results, we calculated the average relative I/O between the compressed and uncompressed R-trees. In addition, we calculated the standard deviation.

The types of queries were:

Point queries. The query region was a random point in the region of the R-tree.

Partial match queries. We specified a value for a subset of the attributes. The query retrieved all tuples that matched the values for the specified attributes. The query was non-discriminatory with respect to the unspecified attributes.

Range queries. We specified a range for each attribute. We varied the hyper-volume of the resulting query region.

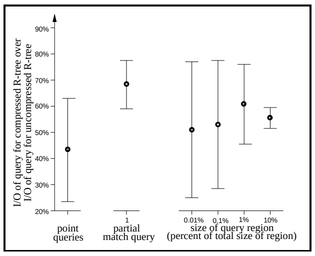


Figure 19: Testing the quality of R-trees on Tiger dataset.

Figure 17 shows the results for the Sales dataset. We used five attributes of that dataset and 750000 tuples. The rate of improvement (page retrieval ratios) is from 35% for point queries to 62% for the worst average on partial match queries. This is a clear win for the quality of the compressed R-tree.

Figure 18 shows the results for the Compustat dataset. Again we used five dimensions and 82600 tuples. In this case we got even better gains than in the Sales dataset case.

Figure 19 shows the results for the Tiger dataset. Here we have two dimensions and 150000 tuples. We see the same good results as in the other datasets.

5 Related Work

Ng and Ravishankar [NR95] discussed a compression scheme that is similar in some respects to our work. In particular, their paper did the following:

- Introduced a page level compression/decompression algorithm for relational data.
- Explored the use of a B-tree sort order over the compressed data as well as using actual B-Trees over the data.

Note, however, that the details of their compression scheme are quite different:

- Our scheme decompresses on a per field, per tuple basis, not a per page basis.
- Except for the actual information in the tuples, we store extra information only on a per page basis. Their compression technique uses run length encoding which stores extra information on a per field per tuple basis.
- Our compression scheme is easily adapted to be lossy, which is important for compressing index pages in R Trees.

Some scenarios that highlight the differences between our schemes are:

- Multiuser workloads that randomly access individual tuples on pages. Clearly, our approach would be superior in this case.
- Performing a range query using a linear scan over the data. Since the bounding box for the entire page is stored on each data page, our scheme can check to see if the entries on the page need to be examined.
- Performing small probes on a B-tree. Using our compression scheme, a binary search can be used to search on a page, since each record is of fixed length. Note that this becomes a serious issue in a compressed environment, where many more entries can fit on a page.

Additionally, we demonstrated the application of multidimensional bulk loading to compression, and presented a range of performance results that strongly argue for the use of compression in a database context.

[EOS81, RH93, Bas85] discuss several compression techniques such as run length encoding, header compression, encoding category values, order preserving compression, Huffman encoding, Lempel-Ziv, differencing, prefix and postfix compression, none of which support random access to tuples within a page.

Like the compression described in Section 4.3, the above techniques, unlike ours, handle any kind of data, but introduce buffer and storage management problems.

[OQ97] discusses several query evaluation algorithms based on the use of compression. While this paper assumes gzip compression is used, our techniques could be used as well in most of the examples discussed there.

6 Conclusions and Future Work

This paper presents a new compression algorithm and demonstrates its effectiveness on relational database pages. Compression ratios of between 3 and 4 to 1 seemed typical on real datasets. Low cardinality datasets in particular produced compression ratios as high as 88 to 1. Decompression costs are surprisingly low. In fact, the CPU cost of decompressing a relation was approximately 1/10 the CPU cost of gunzip over the same relation, while the achieved compression ratios were comparable. This difference in CPU costs means that the CPU decompression cost becomes much less than sequential I/O cost, whereas it was earlier higher than sequential I/O cost. Further, if only a single tuple is required, just that tuple (or even field) can be decompressed at orders of magnitude lower cost than decompressing the entire page. This makes it feasible to store pages in the buffer pool in compressed form; when a tuple on the page is required, it can be extracted very fast. To our knowledge no other compression algorithm allows decompression on a per-tuple basis.

The compression code is localized in the code that manages tuples on individual pages, making it easy to integrate it into an existing DBMS. This, together with the simplifications it offers in keeping compressed pages in the buffer pool (also leading to much better utilization of the buffer pool), makes it attractive from an implementation standpoint. A related point is that by applying it to index pages that contain $\langle key, rid \rangle$ pairs, we can obtain the benefits of techniques for storing $\langle key, rid - list \rangle$ pairs with specialized rid representations that exploit "runs" of rids.

In comparison to techniques like gzip compression, our algorithm has the disadvantage that it compresses only numeric fields (low cardinality fields of other types can be mapped into numeric fields, and in fact, this is often done anyway since it also improves the compression attained by gzip). Note, however, that it can be applied to files containing a combination of numeric and non-numeric fields: it will then achieve compression on just the numeric fields. Nonetheless, the range of applicability is quite broad; as an example, fact tables in data warehouses, which contain the bulk of warehoused data, contain many numeric and low-cardinality fields, and no long text fields.

We also explored the relationship between sorting and compressibility in detail. Among the sorts explored were sorts suitable for bulk loading multidimensional indexing structures and B-trees. The important conclusion is that both sorts worked equally well—which implies that compression will work well on both linear and multidimensional indexes—and are significantly better than no sort. The latter observation underscores the importance of sorting data prior to compression, if it is not already at least approximately sorted.

7 Acknowledgements

We would like to give a warm acknowledgement to Kevin Beyer, for his time as a sounding board, and asking such questions as "Can you support primary indexes?" We would also like to thank Clark French and Sybase for providing us with the details concerning their compression techniques. In addition, we would like to thank Patrick O'Neil for valuable feedback; in addressing his questions, we significantly strengthened this paper.

References

[Bas85] M. A. Bassiouni. Data compression in scientific and statistical databases. In *IEEE Transactions on Software Engineering*, volume SE-11, no. 10, pages 1047–1058, October 1985.

[BKK96] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X—tree: An index structure for high dimensional data. In T. M. Vijayaraman, A. P. Buchman, C. Mohan, and N. L. Sarda, editors, *Proc. 22th Inf. Conf. on VLDB*, pages 28–39, Mumbai, India, September 1996.

[BKSS92] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R^*—tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, 1992.

[BM72] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informat.*, 1:173–189, 1972.

[EOS81] Susan J. Eggers, Frank Olken, and Arie Shoshani. A compression technique for large statistical databases. In *VLDB*, pages 424–434, 1981.

[Gut84] Antonin Guttman. R—trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 47–57, 1984.

[KHS91] H.-P. Kriegel, H. Horn, and M. Schiwietz. The performance of object decomposition techniques for spatial query processing. In *Proc. 2nd Symposium on Large Spatial Databases, Lecture Notes in Computer Science*, volume 525, pages 257–276, 1991.

[Kim96] R. Kimball. *The Data Warehouse Toolkit.* John Wiley and Sons, 1996.

[KSSS89] H.-P. Kriegel, M. Schiwietz, R. Schneider, and B. Seeger. Performance comparison of point and spatial access methods. In *SSD*, pages 89–113, 1989.

[KSSS90] H.-P. Kriegel, M. Schiwietz, R. Schneider, and B. Seeger. The buddy-tree: An efficient and robust method for spatial data base systems. In *Proc. 16th VLDB Conf.*, pages 590–601, 1990.

[LEL96] Scott T. Leutenegger, Jeffrey M. Edington, and Mario A. Lopez. Str: A simple and efficient algorithm for R-tree packing. Technical Report 96-02, Mathematics and Computer Science Dept., University of Denver, 1996.

[LJF94] King-Ip Lin, H. V. Jagadish, and Christos Faloutsos. The TV—tree: An index structure for high-dimensional data. *VLDB journal*, 3(4):517–542, 1994.

[LZ76] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.

[LZ77] A. Lempel and J. Ziv. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 31(3):337–343, 1977.

[NHS88] J. Nievergelt, H. Hinterberger, and S. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. In *Readings in Database Systems.* Morgan Kaufmann, 1988.

[NR95] Wee K. Ng and Chinya V. Ravishankar. Relational database compression using augmented vector quantization. In P.S. Yu and A. L. P. Chen, editors, *IEEE 11'th International Conference on Data Engineering*, pages 540–549, Taipei, Taiwan, March 1995.

[OQ97] Patrick O'Neil and Dallan Quass. Imporved query performance with variant indexes. In *SIGMOD*, pages 38–49, Tucson, AZ, March 1997.

[RH93] Mark A. Roth and Scott J. Van Horn. Database compression. In *SIGMOD Record*, volume 22, no. 3, pages 31–39, September 1993.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+—tree: A dynamic index for multi—dimensional objects. In *Proc. 13th Inf. Conf. on VLDB*, pages 507–518, 1987.

[Sto88] James A. Storer. *Data Compression: Methods and Theory.* Computer Science Press, 1803 Research Blvd. Rockville, Maryland, 1988.