Compressing Relations and Indexes

Jonathan Goldstein Raghu Ramakrishnan Uri Shaft
Computer Sciences Department
University of Wisconsin-Madison

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 and numeric fields. In addition, this new technique supports very fast decompression.

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. 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 (or an entire relation) at a time. Also, all the information needed for tuple decompression resides on the same page with the tuple. This means that a page can be stored in the buffer pool and used in compressed form, simplifying the job of the buffer manager and improving memory utilization.

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. We can also use lossy compression on the internal nodes of an index.

1 Introduction

Traditional compression algorithms such as Lempel-Ziv [10] (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 *(pageid, slotid)* 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 (obviously, query optimization needs to take into consideration the increased performance due to compression). 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.

Performance Study. We implemented all the algorithms presented in the paper, and applied them to various real and synthetic data sets. We measured both compression ratios and processing speed for decompression. Section 4 contains a summary of the results. (A very detailed presentation of the results can be found in [6].)

The performance analysis underscores the importance of compression in the database context. Current systems, in particular Sybase IQ, use a proprietary variant of `gzip`, applied page-at-a-time. (We thank Clark French at Sybase IQ for giving us information about the use of compression in Sybase IQ.) We compare our results to a compression technique similar to the Sybase IQ compression. (We used `gzip` on 64KB blocks). We typically get better compression ratios (as high as 88 to 1). Our compression and decompression techniques are much faster than `gzip`, and are even fast enough for maintaining sequential I/O.

Application to B-trees and R-trees. We study the application of our technique to index structures (e.g., B-trees and R-trees) in Section 3. We compress keys on both the internal (where keys are hyper-rectangles) and leaf pages (where keys are either points or hyper-rectangles). Further, for R-trees we can choose between lossy and lossless compression 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 sort orders can be utilized well for that purpose. We present a bulk loading algorithm that essentially creates a sort order for R-trees. The bulk loading algorithm creates the levels of an indexing structure in bottom up order. If we only create the leaf level, we get a compressed relation. Creating the next levels has little additional cost (typically, no more than 10% the space cost of the leaf level). Thus, the total size of the compressed clustered index is much less than the size of the original relation.

2 Compressing a relation

Our relation compression algorithm has two main components. The first component is called *page level compression.* It 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 fit onto a page, taking advantage of the space freed by compression. Section 2.1 describes page level compression in detail.

The second component of the relation compression algorithm is called *file level compression.* This component takes a list of tuples (e.g., an entire relation) and divides the list into groups s.t. each group can fit on a disk page using page level compression. Section 2.3 describes file level compression in detail.

The most important aspects of our compression technique are:

- Each compressed data page is independent of the other pages. Each tuple in each page can be decompressed based only on information found on the specific page. Tuples, and even single fields, can be decompressed without decompressing the entire page, (let alone the entire relation).
- A compressed tuple can be identified by a page-id and a slot-id in the same way that uncompressed tuples are identified in conventional DBMSs.
- Since tuples can be decompressed independently, we can store compressed pages in the buffer pool, without decompressing them. The way tuple-id's are used does not change with our compression technique. Thus, incorporating our compression technique in an existing DBMS involves changes only to the page level code and to the query optimizer.
- A compressed page can be updated dynamically without looking at any other page. This means that a compressed relation can be updated without using file level compression. However, using file level compression will result in better compression.

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 and 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 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 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)!

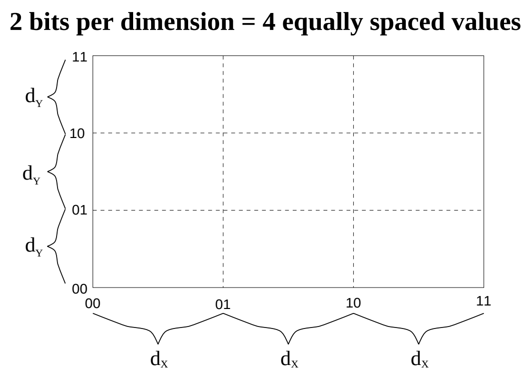

Figure 1: Frame of reference for lossy compression.

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

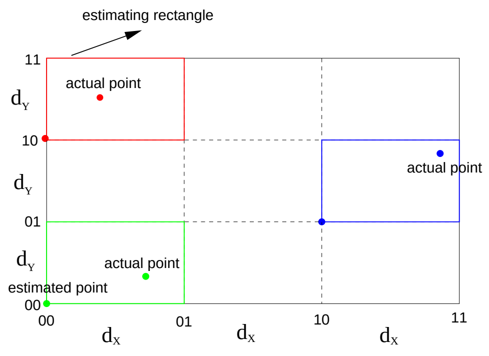

Figure 2: Point approximation in lossy compression.

2.2 Non-numeric attributes

The page level compression technique, as described in Section 2.1, applies only to numeric attributes. However, in some situations we can compress non-numeric attributes. It is common practice in decision support systems (DSS) to identify attributes that have *low cardinality* for special treatment (see [8]). Low cardinality attributes are attributes that have a very limited range of valid values. For example, *gender*, *marital-status*, and *state/country* have very limited ranges although valid values to these attributes are not numeric.

In such systems, it is common practice to map the values to a set of consecutive integers, and use those integers as id's for the actual values. The table containing the mapping of values to integers is a *dimension table*. The *fact table*, which is the largest table in the system, contains the integers. We recommend building such dimension tables for attributes with low and medium cardinality (i.e., up to a few thousands valid values). We get good compression on the fact table, and the dimension tables are small enough to fit in memory or in very few disk pages.

2.3 File level compression

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. For instance, if a database contains 500,000 tuples, there are many ways to group these tuples, and different groupings may yield drastically different compression ratios. [13] demonstrates the effectiveness of using a B-tree sort order to assign tuples to pages. In Section 3 we further develop the connection between index sort orders, including multidimensional indexes like R-trees, and improved compression.

In this section we present an algorithm for grouping tuples into compressed pages. We assume that the tuples are already sorted. The grouping of the tuples maintains the given sort order. The algorithm works as follows:

Input: An ordered list of tuples $t_1, t_2, \ldots, t_n$.

Output: An ordered list of pages containing the compressed tuple. The order of the tuples is maintained (i.e., the tuples in the first page are $\{t_1, t_2, \ldots, t_i\}$ for some i; The tuples in the second page are $\{t_{i+1}, t_{i+2}, \ldots, t_j\}$ for some $j > i$, etc..).

Method: This is a greedy algorithm. We find maximal i s.t. the set $\{t_1, t_2, \ldots, t_i\}$ fits (in compressed form) on a page. We put this set in the first page. Next, we find maximal j s.t. the set $\{t_{i+1}, t_{i+2}, \ldots, t_j\}$ fits on a page. We put this set in the second page. We continue in this way until all tuples are stored in pages.

Note that given the restriction of using our page level compression and the order of tuples, this greedy algorithm achieves optimal compression.

3 Compressing an indexing structure

Many indexing structures, including R-tree variants (e.g., [7, 3]), B-trees [2] , grid files [14], buddy trees [9], TV-trees [12] (using L_∞ metric), and X-trees [4], all consist of collections of *(rectangle,pointer)* pairs (for the internal nodes) and *(point,data)* pairs(for the leaf nodes). Our main observation is: All these indexing structures try to group similar objects (n-dimensional points) on the same page. This means that within a group, the range of values in each dimension should be much smaller than the range of values for the entire data set (or even the range of values in a random group that fits on a page). Hence, 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.

Compressing an indexing structure can yield major benefits in space utilization and in query performance. In some cases, indexing structures take more disk space than any other part of the system. (Some commercial systems store data only in B-trees.) In those cases, the space utilization of indexing structures is important. The performance of I/O bound queries can increase dramatically with compression. If the height of a B-tree is lower, than exact match queries have better performance. In all cases, each page I/O retrieves more data, thus reducing the cost of the query.

Another reason for compression is the quality of the indexing structure. Our work in R-trees yields the following result: As dimensionality (number of attributes) increases, we need to increase the fan-out of the internal nodes to achieve reasonable performance. Compressing index nodes increases the utility of R-trees (and similar structures) by increasing the fan-out.

In Section 3.1 we describe our B-tree compression technique. In Section 3.2 we discuss dynamic and bulk loaded multidimensional indexing structures. Of particular interest is our bulk loading algorithm for compressed rectangle based indexing structures (called GBPack).

3.1 Compressing a B-tree

The objects stored in the B-tree can be either *(key, pointer)* pairs, or entire tuples (i.e., *(key, data)* pairs). The internal nodes of the B-tree contain *(key, pointer)* pairs and one extra pointer. In both cases, we can compress groups of these objects using our page level compression. (The extra pointer in internal nodes can be put in the page header. It can also be paired with a "dummy" key that lies inside the frame of reference.)

3.1.1 Dynamic compressed B-trees.

Note that compressed pages can be updated without considering other pages. However, updates to entries in a compressed page may change the frame of reference. When implementing a dynamic compressed B-tree, we need to observe the following:

- When trying to insert an object into a compressed page, we may need to split the page. In the worst case, the page may split into three pages. (Details on our algorithm for dynamic changes in the frame of reference are in [6].) This may happen when the new object is between the objects on the page (in terms of B-tree order), and the frame of reference changes dramatically because of the new object. (Note that this can happen only if the key has multiple attributes.) The B-tree insertion algorithm should be modified to take care of this case.
- We may not be able to merge two neighboring pages even if the space utilized in these pages amounts to less than one page.
- When deleting entries we can choose to change the frame of reference. However, it is not necessary to do so.

3.1.2 Bulk loading.

We sort the items in B-tree sort order, after concatenating the *key* of each item with the corresponding *pointer* or *data*. We use the file level compression algorithm on these sorted items (see Section 2.3). The resulting sorted list of pages is the leaf level of the B-tree.

We create the upper levels of the B-tree, in a bottom up order. For each level, we create a list of *(key, pointer)* pairs that corresponds to the boundaries of the pages in the level below it. We compress this list using the same file level compression algorithm. The resulting sorted list of pages is another level of the B-tree.

3.2 Compressing a rectangle based indexing structure

Most multidimensional indexing structures are rectangle based (e.g., R-trees [7], X-trees [4], TV-tree [12] etc.). They all share these qualities:

- These are height-balanced hierarchical structures.
- The objects stored in the indexing structure are either points or hyper-rectangles in some n-dimensional space.
- The internal nodes consist of *(rectangle, pointer)* pairs. The pointer points to a node one level below in the tree. The rectangle is a *minimum bounding rectangles* (MBR) of all the objects in the subtree pointed to by the pointer.
- All the MBR's are oriented orthogonally with respect to the (fixed) axes.

In this section we describe our R-tree compression technique. The discussion is valid for the other rectangle based indexing structures as well.

3.2.1 Compression of R-tree nodes.

Our page level compression technique can be modified for groups of *(rectangle, pointer)* pairs. Note that an n-dimensional rectangle can be viewed as a $2n$-dimensional point (i.e., it is represented by minimum and maximum values for each dimension). We can use page level compression on $2n$-dimensional points. We can also save some space by using an n-dimensional frame of reference and treating each rectangle as two n-dimensional points. Note that the pointer part of the pair can be treated as another dimension and be compressed as well.

We make the observation that an R-tree can be used even if the bounding rectangles in the internal nodes are not minimal. In this case, the performance of the indexing structure degrades, but the correctness of search and update algorithms remains intact. In some cases, the leaf level of the tree may contain *(rectangle, pointer)* pairs where the rectangle is a bounding rectangle of some complex object. Again, we may use larger bounding rectangles (i.e., not minimal). In this case, queries may return a superset of the required answers, resulting in some postprocessing. When the minimality of bounding rectangles is not a necessity, we can use lossy compression. There is a tradeoff between degradation of performance due to larger bounding rectangles, and the gain in performance due to better compression.

3.2.2 Dynamic maintenance of the tree.

Since our compression of pages is independent of other pages, we can update the indexing structure dynamically. Similar to compressed B-trees, we need to consider changes in frames of reference due to updates (see [6]). In the R-tree case, splitting a node can result in at most two pages (with B-trees it can result in three). The difference is that the order of objects is not important in an R-tree, so the worst case is realized when the new entry is put in a new page by itself.

3.2.3 GBPack: compression oriented bulk loading for R-trees.

All bulk loading algorithms in this paper partition a set of points or rectangles into pages. The partitioning problem can be described as follows:

Input. A set (or multiset) of points (or rectangles) 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 (or rectangles) that are close to each other in the same group as much as possible. The partition should also be as unbiased as possible with respect to (a set of specified) dimensions.

We bulk-load the R-tree by applying the above problem to each level of the R-tree. We do the bulk loading in a bottom up order. First, the data items (points or rectangles) are partitioned and compressed into pages. Second, we create a set of *(rectangle, pointer)* pairs, each composed of a bounding rectangle of a leaf page and a pointer to that page. We apply the above problem to compress this set. We continue this process until a level fits on a compressed page that becomes the root of the R-tree.

We solve the partition problem by ordering the set of points. Then we apply the packing algorithm (described in Section 2.3). If we have a set of rectangles, we use the ordering of the center points of the rectangles, and then apply the packing algorithm to the rectangles. The most important part is finding a good ordering of the points.

First, we'll give an example of the sorting algorithm. Then, we'll describe it in detail.

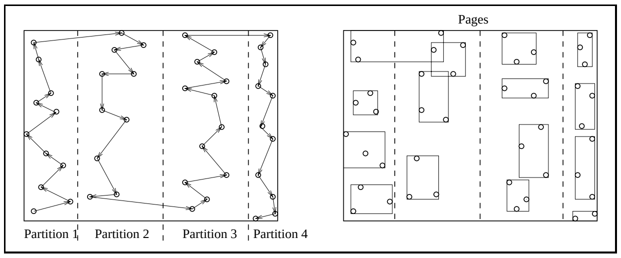

Figure 3: Example for the bulk loading sort operator (Using GBPack).

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 $p := \lceil\sqrt{15}\rceil = 4$ 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.

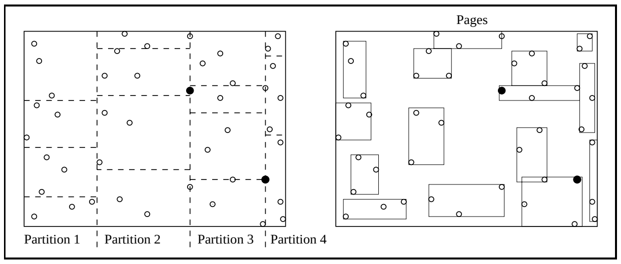

Figure 4: Example for the bulk loading sort operator (Using STR). Bold points belong to partitions to the right of the point.

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 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 [6] 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 the 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 $p := \lfloor P^{1/D} \rfloor$. 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 $sort(A, |A|, D)$. 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.

4 Performance evaluation

This section is a summary of an extensive performance evaluation of our techniques. The full details of the experiments and results are in [6]. These experiments include the compression of many widely varying data sets. When we say that we achieved "40% compression" we mean that the compressed data set is 40% the size of the uncompressed data set. Therefore, lower numbers are better.

Relational compression experiments. We studied both synthetic and real data sets. One real data set (the *companies* relation), consisted of ten attributes taken from a the CompuStat stock market data set, which described companies on the stock exchange. We achieved between 18% compression (for two attributes) to 30% compression (for ten attributes). Another real data set (the *sales* relation) consisted of eleven attributes taken from a catalog sales company. Each tuple described a transaction with one customer. We applied low cardinality mapping (see Section 2.2) and achieved between 10% and 40% compression. A third real data set was the Tiger GIS data set for Orange county, California. We achieved 64% compression with no sorting and 54% with the GBPack algorithm. In all cases, our compression ratios were similar to `gzip`.

When performing experiments on synthetic data sets we observed the following trends:

- The range of values for attributes had a major influence on compression. When the range was small, we needed few bits for representing a value. Therefore, we got better compression.
- Page size had little effect on compression.
- The number of attributes had little effect on compression. We achieved slightly better compression for a lower number of attributes.
- The number of tuples had a logarithmic effect on compression. In general, when the number of tuples doubled, the number of bits needed for representing a tuple decreased by one.
- The data distribution had a major effect on compression. The worst distribution was the uniform distribution. When the data distribution was highly skewed (e.g., exponential distribution) most of the values on a page were in some small range and compression for those values was very high.
- Sorting the tuples increased compression significantly. However, the type of sort order had little influence. B-tree sort order seemed to achieve slightly better compression than GBPack sorting. Therefore, GBPack should be used only when constructing a multidimensional indexing structure.

CPU versus I/O costs. We studied the CPU cost associated with decompressing an entire compressed page. We compared our results to the CPU cost of `gunzip` used in a manner consistent with Sybase IQ. The data set used was the CompuStat relation. Our decompression technique was faster by a factor of between 8 and 20 (depending on the amount of compression achieved). Of particular importance was that our compression technique was fast enough to keep up with sequential I/O on fast disks (we typically achieved 16 Mbytes per second for decompressing all contents of a page). `gunzip` could not keep up with sequential I/O.

Comparison with techniques found in commercial systems. We implemented a compression technique similar to Sybase IQ compression. We used `gzip` and `gunzip` on blocks of data. Each block had size of 64 Kbytes. Note that Sybase IQ uses a propriety compression algorithm that is a variant of the Lempel-Ziv algorithm (`gzip` is another variant of that algorithm). We expect the Sybase IQ algorithm to be more efficient.

On a low cardinality *sales* data set we achieved 1% compression, while Sybase achieved 10%. On medium cardinality *sales* data set we achieved 37% compression while Sybase achieved 23%. On the Compustat data set (which was a mix of low, medium and high cardinality) we achieved 30% compression while Sybase achieved 29%.

Importance of tuple level decompression. Our decompression algorithm can be used for decompressing single tuples and even single fields. If we only need a small amount of information from a compressed page, the CPU throughput increases by orders of magnitude.

For example, consider performing an index nested loops join where the inner relation's index can fit in memory in compressed form. For each tuple from the outer relation we need to probe the index and decompress only the relevant tuples from the inner relation. The cost of this join is dominated by the CPU cost since all the index probes are done in memory. Our decompression can be two to three orders of magnitude faster than `gunzip` in such cases.

Another possible scenario occurs in a multiuser system, when a relation fits in memory in compressed form, but does not fit in memory in uncompressed form. In this case it is possible for many users to access random tuples from the relation.

R-tree compression experiments. The space required by an R-tree or B-tree is slightly larger than the space required for the leaf level of the tree. Therefore, the relational compression results hold for R-trees and B-trees as well. In this section we studied the impact of compression on the performance of R-trees.

We created R-trees for the three real data sets described earlier in this section. We performed point queries, partial match queries (the selection criterion specifies a range for a subset of the attributes) and range queries (the selection criterion specifies a range for all attributes). We measured the amount of I/O needed for several queries on both compressed and uncompressed R-trees.

For the *sales* data set, the average I/O cost using the compressed R-tree was between 35% and 60% the average I/O cost using the uncompressed R-tree. For the CompuStat data set it was between 10% and 60%. For the Tiger data set it was between 45% and 70%.

5 Related work

Ng and Ravishankar [13] 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.

[5, 16, 1] 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. The above techniques, unlike ours, handle any kind of data, but introduce buffer and storage management problems.

[15] 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

[1] M. A. Bassiouni, "Data Compression in Scientific and Statistical Databases", in *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 10, pp. 1047-1058, 1985.

[2] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes", in *Acta Informat.*, Vol. 1, pp. 173-189, 1972.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider and B. Seeger, "The R^*-Tree: An Efficient and Robust Access Method for Points and Rectangles", in *ACM SIGMOD 1992* pp. 322-331.

[4] S. Berchtold, D. A. Keim and H.-P. Kriegel, "The X-Tree: An Index Structure for High Dimensional Data", in *VLDB 1996*, pp. 28-39.

[5] S. J. Eggers, F. Olken and A. Shoshani, "A Compression Technique for Large Statistical Databases", in *VLDB 1981*, pp. 424-434.

[6] J. Goldstein, R. Ramakrishnan and U. Shaft, "Compressing Relations and Indexes", Technical report no. 1366, CS Dept., University of Wisconsin-Madison, December 1997.

[7] Antonin Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching", in *ACM SIGMOD 1984*, pp. 47-57.

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

[9] H.-P. Kriegel et al, "The Buddy-Tree: An Efficient and Robust Method for Spatial Data Base Systems", in *VLDB 1990*, pp. 590-601.

[10] A. Lempel and J. Ziv, "A Universal Algorithm for Sequential Data Compression", in *IEEE Transactions on Information Theory*, Vol. 31, No. 3, pp. 337-343, 1977.

[11] S. T. Leutenegger et al, "STR: A Simple and Efficient Algorithm for R-Tree Packing", Tech. Report, Mathematics and Computer Science Dept., University of Denver, No. 96-02, 1996.

[12] K.-I. Lin, H. V. Jagadish and C. Faloutsos, "The TV-Tree: An Index Structure for High-Dimensional Data", in *VLDB journal*, Vol. 3, No. 4, pp. 517-542, 1994.

[13] W. K. Ng, C. V. Ravishankar, "Relational Database Compression Using Augmented Vector Quantization", in *IEEE 11'th International Conference on Data Engineering*, pp. 540-549, 1995.

[14] J. Nievergelt, H. Hinterberger and S. C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure", *Readings in Database Systems*, Morgan Kaufmann, 1988.

[15] P. O'Neil and D. Quass, "Improved Query Performance with Variant Indexes", in *ACM SIGMOD 1997*, pp. 38-49.

[16] M. A. Roth and S. J. Van Horn, "Database Compression", in *SIGMOD Record*, Vol. 22, No. 3, pp. 31-39, 1993.