Indexing and Compression in Data Warehouses

Kiran Goyal*, Krithi Ramamritham†, Anindya Datta‡, and Helen Thomas§

Abstract

Efficient query processing is critical in a data warehouse environment because the warehouse is very large, queries are often ad-hoc and complex, and decision support applications typically require interactive response times. Existing approaches often use indexes to speed up such queries. However, the addition of index structures can significantly increase storage costs. In this paper, we consider the application of compression techniques to data warehouses. In particular, we examine a recently proposed access structure for warehouses known as *DataIndexes*, and discuss the application of several well-known compression methods to this approach. We also include a brief performance analysis, which indicates that the DataIndexing approach is well-suited to compression techniques in many cases.

1 Introduction

Data warehouses are large, special-purpose databases that contain historical data integrated from a number of independent sources, supporting users who wish to analyze the data for trends and anomalies. The process of analysis is usually done by queries that aggregate, select and group the data in a number of ways. Efficient query processing is critical because the data warehouse is very large, queries are often complex, and decision support applications typically require interactive response times.

A highly normalized schema offers superior performance and efficient storage for OLTP where only a few records are accessed by a particular transaction. The star schema provides similar benefits for data warehouses where most queries aggregate a large amount of data. A star schema consists of a large central fact table which has predominantly numeric data and several smaller dimension tables which are all related to the fact table by a foreign key relationship. The dimension tables have more descriptive attributes. Such a schema is an intuitive way to represent the multi-dimensional data so typical of business, in a relational system. The queries usually filter rows based on dimensional attributes and then group by some dimensional attributes and aggregate the attributes of the fact table.

There has been some research on creating a set of summary tables to efficiently evaluate an expected set of queries. This approach of materializing needed aggregates is possible only when the expected set of queries is known in advance. But, when ad-hoc queries must be issued that filter rows by selection criteria that are not part of the dimensional scheme, summary tables that do not foresee such filtering are useless. Also, precomputing all necessary tables is often infeasible, as this grows exponentially with the number of dimensions. In such cases queries must be evaluated using indexes to efficiently access base data.

*Indian Institute of Technology, Bombay
†Indian Institute of Technology, Bombay and Univ. of Mass. Amherst
‡Georgia Institute of Tech.
§Georgia Institute of Tech.

Data warehouses are updated only periodically, and during this time no queries are allowed. This allows the batch update process to reorganize the indexes to an optimal form, in a way that would not have been possible otherwise. Since the problems of maintaining indexes in the presence of concurrent updates is no longer an issue, it is possible to use more specialized indexes to speed up query evaluation. In this paper we concentrate on this approach.

So far, indexes have generally been considered separate from the base data. Given the large size of data warehouses, storage costs are very high and so is the cost of storage due to index structures. DataIndexes provide indexing at no extra cost apart from that of storing the data, thus saving a lot of space. We propose compressing the data in the form of DataIndexes. Compression gives dual benefit. Not only does it reduce storage costs, but also results in efficient query processing by reducing the number of disk accesses [RH95]. Although decompression adds to processing cost, the [RH95] paper showed that in databases the reduced number of disk accesses more than compensates for the increased processing time. We believe that it should in fact be more efficient to compress a data warehouse as most queries access a large amount of data and most of the decompressed data will actually be required to evaluate the query. In this paper we explore how compression performs in a data warehousing environment.

The remainder of this paper is organized as follows. In Section 2 we describe several indexing approaches used in data warehousing and in Section 3 we review various compression techniques. In Section 4 we discuss how compression may be applied to certain types of indexes and in Section 5 we conclude the paper.

2 Variant Indexes

In this chapter we describe some of the variant indexes reported in previous work [PQ97], including the new DataIndexes [KA98]. We also describe some of the algorithms to use with these indices.

2.1 Bitmap Indexes

Most database systems today use B-trees as indexes. This uses a tree ordering and has an entry for each key value which references all rows in the table having that key value. Each reference is a unique Row ID, which specifies the disk position. Such a Row ID may occupy 4 to 6 bytes. In OLTP applications a particular entry has very few references compared to the total number of rows in the table. However, in OLAP each entry may have several references. In such a case, it is more efficient to represent the list as a bitmap. We take a bitmap of length equal to the number of rows in a particular table and a one at ordinal position i indicates that the i^{th} row is present in the list. Thus, we have just a new way of representing the list of RowIDs and such an index is called a Bitmap Index. The following example illustrates why these are more efficient in an OLAP environment. Consider a column Gender having only two possible values Male or Female. In this case the entries for both key values will have about half the total number of rows. A Row ID representation will occupy $4 * n$ bytes in all while a bitmap representation will occupy only $2 * n/8$ bytes where n is the total number of rows in the table. This results in enormous disk savings especially if n is large. There is disk saving as long as average density of a bitmap is greater than 1/No of bits in a RowID. Also Bitmaps are a lot more CPU efficient than RowIDs. The common operations of taking union and intersecting RowID lists take several instructions, while most processors have single instructions to directly AND, OR and NOT machine words. Thus, Bitmap operations are a lot faster. When bitmaps are sparse one can compress them. The simplest of compression methods is to convert it back to a RID list.

A further optimization described in [PQ97] is to divide the table into segments of a fixed number of rows each. The bitmaps corresponding to each segment is a fragment and each fragment can be stored as a bitmap or an RID list. This not only saves space, but also saves disk I/O in combining bitmaps. This is because some fragments may not have any set bits at all and hence while ANDing, the other bitmap may not be retrieved at all. This is particularly useful for clustered data.

[CY98] draws a parallel between bitmap indexing and number systems. The idea is that when you orient bitmaps vertically and place them side by side you can think of each entry being represented in base n where n is the number of distinct attributes. They represent it in different bases and discuss space time tradeoffs. In particular, base 2 representation gives bitslice indexes. [WB98] describes methods of efficiently coding Bitslice indexes so that the least number of bitmaps have to be accessed for range selections.

2.2 Projection Index

A projection index on a column C consists of stored column values from C in the same order as they appear in the table T from which they are extracted. It is easy to see that there is a simple arithmetic mapping to get the value corresponding to a given ordinal number provided the column is fixed length. However, if the column is variable length, we can either fix a maximum size for the column or use a B-Tree to access values using the row number. The Projection index is very useful when one has to retrieve column values corresponding to a very dense FoundSet (A bitmap having a set bit for every row to be retrieved). This is because more values fit onto a page, and a single read may bring in more than one value in the FoundSet. OLAP queries have very low selectivity queries and retrieve only a few of the columns. Thus, such an index is extremely useful.

2.3 Join Index

A join index by definition is an index on one table for a quantity that involves a column value of another table through a common join. Most Join Indexes represent the fully precomputed join in different ways.

The Star Join Index concatenates column values from different dimensional tables and lists RIDs in the fact table for each concatenated value. The problem with this index is that they are not efficiently recombinant and an index has to be made for every combination of concatenated values. Bitmapped Join Indexes solve this problem. It consists of using a bitmap index that uses RIDs of a dimensional table instead of search key values to index the records of the Fact table. Thus, to compute a query one simply ORs all bitmaps for selections on each table and then ANDs all the resultant bitmaps for each of the Dimension tables. It is clear that only one index is necessary for every dimension table.

2.4 GroupSet Index

This index improves the efficiency of grouping queries by creating a special index and clustering the data. First, the dimensional tables must be ordered according to a primary key or a hierarchy which ever is more appropriate depending on the queries a user would want. The fact table rows are then clustered so that rows with the same foreign keys on all dimensions are clustered together on disk. Further, the successive groups are in the same order as a nested loop on the values in the dimension tables. In other words, the fact table is sorted based on the ordinal position of the rows in the dimension tables which join with the particular fact table row. A B-Tree Index is then built using the concatenation of the dimension key values and using the same ordering among them as in the groups of the fact tables. For each concatenated key value the ordinal number of the starting of the corresponding group in the fact table is stored. The end of a group is simply one before the beginning of the next group. It is easy to see that a group by can be performed in a single pass of the fact table. Also groups not having any representative rows in the fact table are not stored in the index. Thus, the Groupset Index is efficient for aggregation. Also, only an integer is stored for each key value which is quite space efficient.

2.5 DataIndexes

DataIndexes [KA98] are a new way to represent the data in a warehouse in which the Indexing is essentially free in terms of storage. The rest of the paper assumes we have the data in the form of DataIndexes. An assumption made in DataIndexes is that referential integrity is strictly enforced on all star schemas. In this section, we describe the two types of DataIndexes and algorithms to evaluate queries using these Indexes.

2.5.1 Basic DataIndex (BDI)

As we saw earlier projection indexes are particularly suited for OLAP queries. Also notice that it is very easy to form the original table if we have projection indexes on each of the columns. Thus, if we force projection Indexes on all columns one might as well do away with the original data as it is redundant. This is the main idea behind the BDI. The BDI generalizes projection indexes by allowing any number of columns to be in a single BDI. It would be useful to store two or more columns in a single BDI if it is known that most queries will access them together. Thus, the BDI is a vertical partition of the table where each partition may have one or more columns. Clearly, the Projection indexing is free in terms of storage. A graphical representation of the BDI is shown in Figure 1. In this figure the original table of four columns is stored as two BDIs, one of one column and the other of 3 columns. The dotted lines show the ordinal mapping between them.

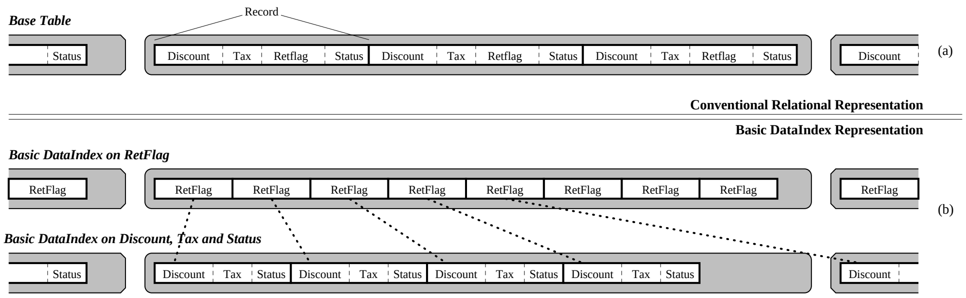

Figure 1: The Basic DataIndex

2.5.2 Join DataIndex (JDI)

If we store the foreign key column also using BDIs, we must join it every time with the corresponding column of the dimension table. The BDIs rely on ordinal mapping. Thus, it is a good idea to store the ordinal number of the row it references in the foreign key column. This is called a JDI. In a way this precomputes the join. Thus it is a join index. This too is free in terms of storage as one does not need to store the column value any more. In fact, we may take lesser storage if the width of the key column is larger than the maximum ordinal number. Such a JDI is shown in Figure 2. The sales table is the fact table having Timekey as foreign key referencing the Time Dimension Table. The figure uses dotted lines to show as in BDIs the ordering according to row number, while solid lines are used to show the rows to which different entries in the JDI point. Using such a representation, joins may be performed very efficiently (in a single pass). There are two algorithms depending on the amount of memory you have. The SJL (Star join with large memory) and the SJS (Star Join with small memory) are discussed next.

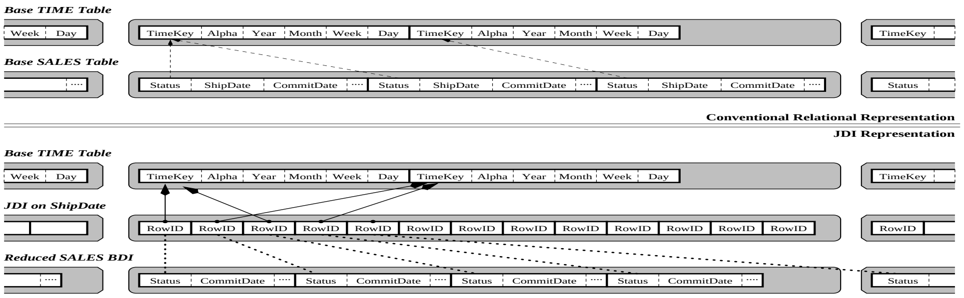

Figure 2: The Join DataIndex

2.5.3 Star Join with Large Memory (SJL)

The SJL uses the JDI to do a star join in a single pass of the relevant BDIs of the fact table. The algorithm is shown in Algorithm 1.

Let D be the set of dimension tables participating in the join, F be the fact table, C_D be the set of columns of dimensional tables that contribute to the join result and let C_F be the fact table columns that contribute to the join result. To answer the query first we form rowsets (a bitmap having the i^{th} bit checked to denote that the i^{th} row if the table is selected) $R_1 \ldots R_{|D|}$ on each of the dimension tables and R_F for the Fact table. This can be done by a scan of the appropriate BDIs. The SJL algorithm starts by loading all BDIs corresponding to columns in C_D (Steps 1 to 4). If there is a predicate in the query on any of these columns one may load only the blocks that have some row in the corresponding rowset. Now, for all records corresponding to the rowset R_F we scan all JDIs on a table belonging to D and check whether the row pointed to by the JDI is there in the corresponding rowset. This is done by steps 6 to 8. If this check succeeds for every JDI then the output record is built by using the in memory dimensional BDIs and reading the corresponding row of fact table BDIs having columns in C_F. This is done by steps 9 to 12. Clearly, this algorithm is very efficient and difficult to improve upon for the dense rowsets of typical OLAP queries. However, it requires all BDIs having columns in C_D to be stored in memory. Some dimension tables may be huge and such large amounts of memory need not be available. Thus, we discuss another algorithm that needs very little memory next.

Algorithm 1: The SJL Algorithm

Note: Enough memory to hold all BDIs in the join result
Input:
  D: Set of dimensional tables involved in the join
  C_D: Set of dimensional table columns that contribute to join result
  C_F: Set of fact-table columns that contribute to the join result
  R: Set of rowsets for each table in D and the fact table F. These were computed before SJL starts
01: for each column $c_i \in C_D$ do
02:     for each row $r \in R_i$ do
03:         if the block of BDI on c_i where r is located is not loaded then
04:             load this block in to memory array a_i
05: for each row $r \in R_F$ do
06:     for each JDI j on a table of D do
07:         if $j(r) \notin R_j$ then
08:             goto 5
09:     for each $c_i \in C_D$ do
10:         $s[c_i] \leftarrow a_{c_i}(j_{c_i}(r))$
11:     for each $c_i \in C_F$ do
12:         $s[c_i] \leftarrow r[c_j]$
13:     Output s

2.5.4 Star Join with Small Memory (SJS)

The SJS does the star join in whatever little memory the system has. The algorithm is shown in Algorithm 2. We use the same symbols as in SJL and the algorithm assumes that rowsets on all tables for all restrictions are already provided in memory.

The algorithm has four phases. The first phase (steps 1 to 4) restricts R_F to only those rows which will appear in the join result. The next phase (steps 5 to 7) write only those parts of the relevant JDIs that correspond to the restricted rowset R_F. These two phases have basically reduced the amount of data we have to handle so that we may make multiple passes over the data. In the third phase(steps 8 to 19) the actual join takes place. Since memory may be insufficient to load all BDIs needed in the result, we load as many blocks of the BDIs as can be loaded. Then, we scan the JDIs for ordinal numbers that correspond to some row that is in memory. If we get a matching entry in memory it is stored into temporary BDIs along with a rowset indicating which rows are being stored. This is done till all BDIs have been loaded in memory once. Thereafter, all the temporary BDIs are merged using the rowsets stored along with them in step 19. The fourth phase (steps 20 to 25) simply assembles the output record using the temporary merged BDIs and the fact table. It is clear that this is a lot more expensive compared to SJL. In particular the dimensional table columns may be wide and storing intermediate results containing the join of such a column with the fact table and then doing a merge may be quite expensive. We have attempted to solve this problem under some conditions in the Section 2.5.5.

Algorithm 2: The SJS Algorithm

Note: Negligible Memory requirements
Input: Same as in SJL
for each JDI j on a table of D do
    for each row $r \in R_F$ do
        if $j(r) \notin R_j$ then
            $R_F \leftarrow R_F - \{r\}$
for each JDI j_t on a table $t \in D$ do
    for each row $r \in R_F$ do
        write $j_t(r)$ to temporary JDI $j_{t,temp}$ on disk
for each BDI b_i on a column $c_i \in C_D$ do
    $k \leftarrow 1$
    while $\exists$ unloaded blocks of b_i do
        Create temporary BDI $b_{i,k}$ on disk
        Create temporary rowset $R_{i,k}$ on disk
        Load as many blocks of b_i as possible into in-memory array a_i
        for each row r in $j_{t,temp}$ do
            if $b_i(r) \in a_i$ then
                $b_{i,k} \leftarrow b_{i,k} \cup b_i(r)$
                $R_{i,k} \leftarrow R_{i,k} \cup r$
        $k \leftarrow k + 1$
    Merge all $b'_{i,k}s$ (using rowsets)
for each row $r \in R_F$ do
    for each column $c_i \in C_D$ do
        $s[c_i] \leftarrow b_i(r)$
    for each column $c_j \in C_F$ do
        $s[c_j] \leftarrow r[c_j]$
    Output s

2.5.5 Horizontal Partitioning

We now describe a scheme that will reduce the memory requirements of SJL and thus provide performance advantages in many cases. The scheme is to divide the larger dimension tables into horizontal partitions, the size of which is such that it fits comfortably into memory along with some of the partitions of the other similar large dimension tables. This may sound vague but it will become clearer later that this can be decided given the schema and a query profile. Next the Fact table is organized such that records referencing rows of a particular combination of partitions of the partitioned dimension tables are stored contiguously in separate files. A similar technique is also done for caching in [ND97].

Now, one can do a join in a manner very similar to the SJL, but requiring less memory. The algorithm is to get a combination of partitions of the partitioned dimensions into memory and then scan the JDIs in the file corresponding to this particular combination of partitions. Once this is done, we load a new combination and repeat the same procedure. The new combination is chosen in a gray code order in the sense that only one of the partitions is replaced. One such order is a nested loop order in which we go forward and backward alternately on each of the dimensions. Clearly we have avoided repeated scans of the fact table and also the merge of the intermediate results as in case of SJS. However, we may scan some parts of some BDIs several times. The number is multiplicative in the number of partitions of the BDIs which are higher than this one in the nested loop order. In other words, if $n_1, n_2, \ldots, n_k$ are the number of partitions of BDIs $B_1, B_2, \ldots, B_k$ then column of BDI B_1 is scanned once, B_2 is scanned B_1 times, B_k is scanned $B_1 * B_2 * \ldots * B_{k-1}$ times. Thus, if we have additional memory, we may treat more than 1 partition of the BDI B_k as a single partition and gain in performance. Given this, it might seem that making finer and finer chunks would do no harm. This is not the case. As we shall see later, compression using our schemes suffers as you horizontally partition the data. The partitions are indexed and as the number of partitions increase in number, indexing overheads also increase. Also, if memory is very small and the fact table is not much bigger than these dimensional tables and the query involves many of these large dimensional tables, then SJS may even be better.

We may get several additional advantages out of this scheme.

1. The partitioning of dimension tables may be done after arranging them in order of either hierarchy as is commonly observed in data warehouses or ordered by a column on which range queries are common. This way a query will usually involve only one or few of the partitions of such a dimension table. The performance gain out of this should be obvious as we have saved going through a significant part of the fact table. Another minor benefit possible is that since now the partitioned JDIs store ordinal numbers from a smaller domain, we can store only the difference between each value and the smallest value achieving something like prefix compression. However, this is not going to be significant if the number of partitions of the corresponding dimension table is small.

2. Also, data is clustered though only to a limited extent. Thus, we have better chances of reading more than one value that contributes to the join result in one block.

3. There is a parallel version of the SJL[KP98]. This algorithm requires (to be efficient) that processors have enough memory to hold entire BDIs. Horizontal partitioning would be invaluable in this case.

3 Compression

As we saw in the introduction compression has benefits apart from the obvious savings in disk space. Firstly, query processing may improve, as lesser number of blocks have to be retrieved. This is because more rows now fit into a page and a single read may bring in more tuples required for the query. Secondly, the effective memory size is increased since more tuples may be stored in the compressed form. Also, network communication costs decrease in the case of distributed databases.

The main problems with compression include the high decompression cost at CPU and the fact that attributes become of variable length. Thus, we must compress in a way that the overheads due to these are more than compensated by the benefits discussed previously. To this end, let us first see what are the types of redundancy that most compression algorithms use.

There are four commonly encountered types of redundancy.

- **Character Distribution:** Some characters may be used more frequently than the others. Specifically, in 8 bit ASCII representations nearly three quarters of the 256 characters may never be used in most text. Also the letter e and 'space' are most common in English text.
- **Character repetition :** Strings having repetitions of a character or a set of characters may be more compactly represented by giving the character and the number of times it is repeated. Run Length Encoding (RLE) does precisely this. Such repetitions commonly occur in the form of spaces in text or when a variable length column is represented as a fixed length column of maximum width.
- **High usage patterns :** Certain sequences of characters may occur more frequently than others. For example, a period followed by a space is much more common than any other pair of characters. Also words like compression and index are probably the most common words in this paper.
- **Positional Redundancy :** Certain characters consistently appear at a predictable place in the data. For example, some columns like 'special handling' may mostly contain a NULL value.

A discussion of the common techniques of compression exploiting these types of redundancy follows.

3.1 Compression Techniques

Most compression techniques are based on either the *dictionary model* or on the *statistical model.* The dictionary based schemes maintain a dictionary of commonly used sequences and output shorter codes for them whenever they are encountered. Statistical schemes compress by assigning shorter codes to more frequent characters. Another dimension of compression is that they may be *adaptive* or *non-adaptive.* Adaptive schemes don't assume anything about the data but tune their model as they compress, while non-adaptive schemes assume that the data will have some properties. A short description of some popular compression algorithms follows.

3.1.1 Huffman Coding

Huffman coding uses the fact that not all characters in most meaningful data occur with the same frequency. Some characters are more frequent than the others. In Huffman coding [HD52], each character is assigned a code of integral number of bits such that the length of the code is closest to log_2(probability of symbol). Thus, frequently occurring characters are assigned shorter codes. Huffman was also able to prove that his method cannot be improved upon by any integral bit width coding stream. Note that each code has a unique prefix to enable correct decoding despite the variable length of the codes.

Here is an illustration of how the codes are assigned given the frequencies of each of the characters. Suppose there are only 5 characters A, B, C, D and E with the frequencies 15, 7, 6, 6 and 5 respectively. First, the individual symbols are laid out as a string of leaf nodes that are going to be connected by a binary tree. Each node has a weight which is equal to the frequency of the symbol. Then the following steps are done to make the tree in Figure 3.

- The two free nodes with the lowest weights are located.
- A parent node for these two is created with weight equal to the sum of the two child nodes.
- The parent node is added to the list of free nodes and the two children are removed.
- One of the child nodes is designated as the path taken from the parent node when decoding a 0 bit, while the other is set to 1.
- The previous steps are repeated until one free node is left which is made the ROOT.

There are both adaptive and non-adaptive flavors of Huffman coding. The non-adaptive flavor has two passes. In the first pass it scans or samples the data and gets the frequencies of the characters, while in the second it actually compresses with the tree built statically. The adaptive flavor has only one pass. Initially, it assumes that all characters are equally probable, but it keeps updating its model as it learns more about the data.

Another good feature of Huffman coding is that it is efficiently implementable in hardware, i.e. it can be implemented on a VLSI chip [RS91] that can encode and decode at speeds as fast as the disk transfer speeds. The result is absolutely no compression overhead. However, these chips are not yet manufactured and would still be very expensive.

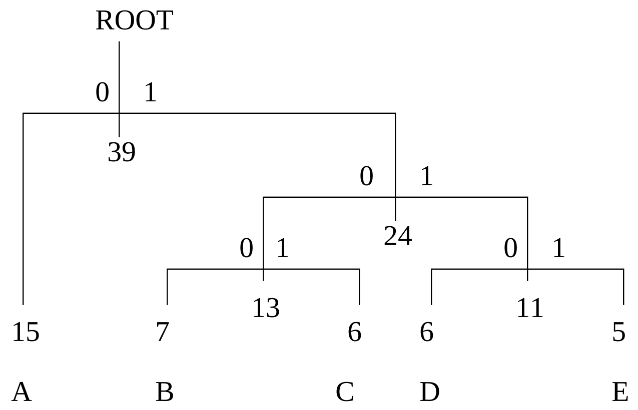

Figure 3: Huffman Coding

3.1.2 Arithmetic Coding

Arithmetic coding [WC87] improves on Huffman coding by removing the restriction that integral number of bits be used. It actually represents each character in fractional number of bits. The compressed data is represented as an interval between two reals. Initially, that range is [0,1). Each character is assigned a interval in the range [0,1) and the width of which is proportional to its probability of occurrence. For example, if there were only 4 characters a, e, i and u and they occurred with frequencies in the ration of 2:3:1:4 then a possible assignment is shown in Figure 4. When a character appears the range is narrowed to the fraction assigned to that character. In the figure, when e is encountered the range is narrowed to [0.2,0.5) and then similarly when a is encountered next. As the string is encoded the range progressively decreases and the number of bits needed to specify any number in that interval increases. In the end, the number in the interval requiring the least number of bits is output as the compressed form. In the figure, the final range is [0.2,0.26) and hence 0.25 requiring only 2 bits is output.

There are adaptive and non-adaptive methods for arithmetic encoding too, and the differences are very similar to the Huffman coding described previously. The implementation details are much more complex as it involves doing infinite precision arithmetic in a finite precision computer and also outputting incrementally. One does not want to keep the entire compressed number in memory and output it only at the end. Another serious problem is that the implementation involves too many divisions and multiplications which makes it slow.

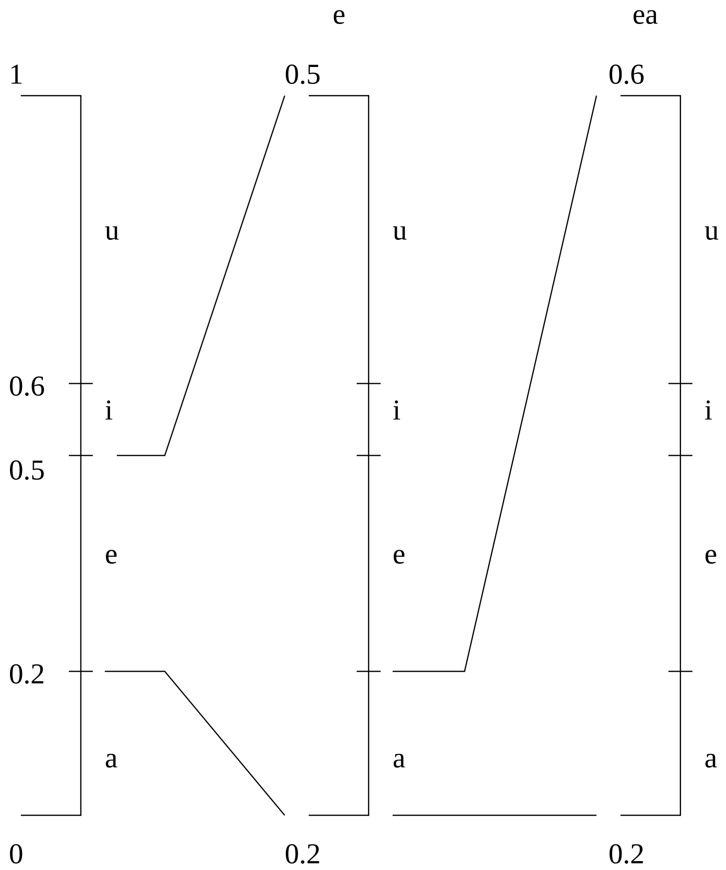

Figure 4: Arithmetic Coding

3.1.3 LZW

The LZW [We84] compression method follows the dictionary model. The scheme involves a finite string translation table (usually having a size of around 8K to 32K) which has strings and their corresponding fixed length code values. The table is initialized with all possible characters. At each step, it searches for the longest match of the current string in the table and outputs the appropriate code for it. It then extends the string by the next character in the text and adds it to the dictionary. Thus, the table has a property that if a particular string is in the table all its prefixes are also present. Thus, the table is built adaptively and does particularly well at text. Though it is meant to be adaptive, there is an obvious non-adaptive version that does not perform as well. Simply build a full dictionary in a sampling phase and then compress it in a second scan without updating the dictionary. Commercial programs like "Compress" discard a full dictionary as soon as the compression ratio begins to decrease significantly.

Clearly, the compression and decompression algorithms have to maintain the dictionary and do equal amount of processing. Both maintain a tree. The compression algorithm uses a hash function to locate the child while the parents are explicitly maintained in each node. Thus, compression and decompression are equally fast.

3.1.4 LZ77

This is also a dictionary based technique. It uses a window into the previously seen text as its dictionary. The main data structure is a text window, divided into two parts. The first consists of a large block of recently decoded text. The second, which is much smaller, is a look-ahead buffer having characters not yet encoded. The algorithm tries to match the contents of the look-ahead buffer to a string in the dictionary. At each step it tries to locate the longest match in the dictionary and codes the position of the match and the length of the match. If there is no match the character is simply output as it is. To differentiate between the two types of tokens, a single bit prefix is added. An example of the window is shown in Figure 5, where the next string will be coded as pointer to the phrase 'MAX' in the dictionary denoted by a pointer in the figure and the length will be four.

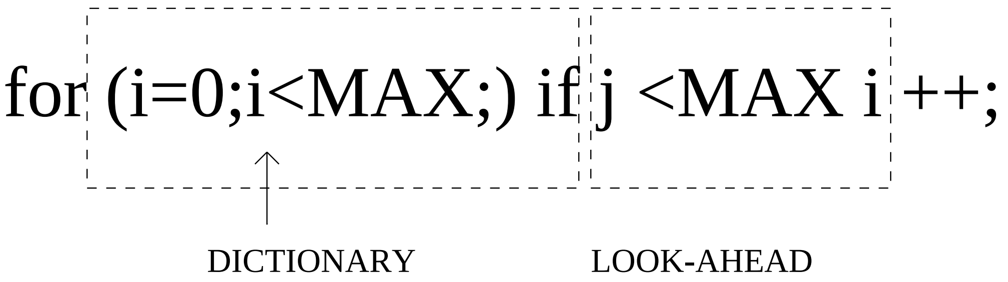

Figure 5: LZ77

An interesting feature of the algorithm is the way it handles repetitions of sequences of characters. For example consider the case of Nrepetitions of a single character 'A'. It will add the first character as it is and then encode the remaining as a pointer to the previous 'A' and the length of the match as $N-1$, provided the length is less than the look-ahead size. It is easy to see how the decompresser can expand such a code. It will simply keep copying the character at the k^{th} location to the $(k+1)^{th}$ location. Clearly, this method is inherently adaptive and the non-adaptive version of the algorithm does not make sense.

3.1.5 Block Sorting

This is a very recent compression technique [BW94] and performs extremely well on textual files. It exploits the fact that characters immediately following a particular character in English text are from a reduced domain. Even if several characters are there in that domain some characters are much more frequent than the others. For example, 'e' will often follow 'h' as in 'the' which is a very frequent word. However, 'z' will almost never follow any consonant.

What it does is to first transform the data to a form in which it is compressible by other algorithms. The transformation is as follows. First, all cyclic permutations of the text string are sorted. The position of the original string is noted along with the last characters of the sorted permutations in that order. It might not be very obvious so as to how this can be reversed efficiently but you can look into the references given above. Now the characters are clustered, and a locally adaptive technique such as move-to-front coding is applied. This starts with the character set in a list in some fixed order. Then, as a character comes, we output its current position in the list and move it to the front of the list. Clearly, since the characters are clustered together, the lower numbers like 0,1,2 will dominate. This is a perfect opportunity for arithmetic coding. This is only a conceptual overview and there is a lot more in implementing it. This scheme achieves compression comparable to the best statistical compressors for English text in much lesser time.

4 Compression of DataIndexes

We now turn our attention to applying some of the indexing and compression concepts to the data in a data warehouse. We feel that data stored in the form of DataIndexes is much more compressible than storing it as tuples in a table. This is primarily because different columns have different distributions and different methods could be applied to different columns. In the following sections, we show schemes for compression of fact table BDIs, JDIs and the Dimension tables.

4.1 Compressing Text Columns of the Fact table

In this section, we shall figure out which algorithm to use for compressing the text columns of the fact table. Note that apart from the algorithm compressing well, it should also do it at a good speed. Another aspect of compression is the granularity at which it is done. We can do it at the attribute level or at the block level. By block level, it is meant that you compress as much as possible in to a single disk block, and then restart compressing in a new block. This ensures that each block can be decompressed independently of all the others. Care should be taken so that no attribute is between blocks, so that no attribute causes more than a single I/O.

To begin with, let us see how various algorithms perform relatively at an average in terms of compression size, compression speed, as well as expansion speed on normal English text. To measure how well an

Algorithm	Compression Ratio	Compression Speed	Expansion Speed
Huffman	40	14	12
Arithmetic	41	4.5	2
LZ77	60	3.5	17
LZW	56	13	13

Table 1: Comparison of Compression Algorithms

algorithm compresses data we define Compression Ratio (CR) to be $1-$(Compressed size/original size). The numbers for Compression and Expansion speed are proportional to the amount of data they can handle in unit time.

From Table 1, a number of things may be concluded:

1. Clearly, Arithmetic Coding is too slow in terms of compression and expansion speed compared to Huffman Coding and provides little gain in compression. Thus, Arithmetic Coding should not be done in a database where performance is an issue.

2. For text files, the dictionary based techniques perform substantially better than statistical techniques.

3. Expansion Speed is particularly critical in a Data Warehouse, as data will be queried and hence expanded several times while it will be compressed very few times when data is added or reorganized. Therefore, the LZ77 method seems to have an edge over LZW.

It should be clear that LZ77 and LZW are the only choices we have. Let us consider them one by one at the attribute level as well as the block level. At the block level, both the algorithms perform similarly, though there is some loss in compression ratio for both; however the difference between their compression ratios is not really significant. Thus, we must choose LZ77 unless updates are huge and very frequent. If there are very frequent updates or reorganizations, compression speed may become an issue and then it is better to use LZW as it compresses also at a good speed.

At the attribute level, things are quite different. We do not expect long sequences of characters being repeated within a single column value which itself is of a few tens of bytes. Thus, after going through the LZ77 algorithm it is clear that it will not do well at the attribute level. It might even result in a slight expansion. The adaptive flavor of LZW will also do equally badly for the same reason. One cannot expect to have built a tree that results in substantial compression by going through such a small amount of text. Now, consider the non-adaptive version of LZW. For the non-adaptive version of LZW at attribute level, nothing much has changed as compared to file level. In both cases we sample data and the dictionary used in both cases will be very similar. The only difference is that at the attribute level we cannot over-step attribute boundaries while compressing the data. Also, as pointed out in [IW94], non-adaptive LZW at record level performs better than Huffman coding for their datasets. However, one has to sample carefully and this does well only when most values will consist of words from a very selective range. Long English documents are not compressed by much using non-adaptive LZW. Thus, non-adaptive LZW appears to be the best choice for the attribute level.

A Sample Column
0070000
0080595
0168700
0043290
0001355
0087795
0009995

Table 2: Redundancy in Numeric Columns

4.2 Compressing Numerical columns

The most often queried columns in the fact table are the measures which are mostly numerical. These are the ones which are mostly aggregated in Warehouse queries.

First, consider compressing them at the attribute level. It is reasonable to assume that most fields will occupy 4 bytes(capable of representing more than $4 * 10^{30}$) or less. Such numbers, represented in binary do not compress by as much as text, and most good compressors compress it to around 50% of the original size. This means that a column value will be represented in 2 to 3 bytes, rounding to the next higher number. Add to this, another byte to store a pointer to the attribute as the data will become variable length and then each compressed attribute occupies 3 to 4 bytes. This results in compression ratio of 0% to 25%. Clearly, compressing these at the attribute level does not make sense. Thus, if many of the queries require few attributes from each block, it is not a good idea to compress the numerical columns.

Let us now concentrate on compressing at the block level. First, we must find out what we can make use of in compressing numerical data. If the numbers are totally random uniformly distributed over the range that can be represented in that width, then there is no redundancy and no algorithm will compress it. It will be best to store such data without compressing it. However, in real life warehouses, some columns by their very nature will have most values in the lower and middle range or most will be multiples of 5, 10 or 100. To illustrate what we mean, we have constructed an example column in Table 2. A column of balances in a bank is likely to be such a column. The width of the column will be large, as there can be very large balances, though rarely. In other words, if such a column is actually 8 bytes in width, most values will use only 4 bytes. Statistical techniques like arithmetic coding look appropriate for such data.

Note that the major skew in the frequency of digits, which is exploited by statistical coding techniques, is in the higher order bits and the lower order bits. Besides, they have different kinds of skews. Thus, instead of having one common frequency table, it will be better to have a different frequency table for every byte of the width of the column at the price of a small decrease in speed. Since the width is small, overhead of storing the frequency distributions is not much and compression efficiency should improve by quite a lot. Here is an example to show how having different frequency tables may be beneficial. In this example we will use statistics at the bit level since computing for ten sets is too much and the simple case is sufficient to illustrate the concept. Assume a 32 bit attribute. Let 90% of the bits in the most significant 8 bits be 0's. Assume an equal distribution in the lower 24 bits. Arithmetic coding compresses to a maximum of $\sum -p_i log_2 p_i$ [WC87] where i denotes the character. For this example the bits are the characters. Therefore, it can be compressed to $-0.9log_2 0.9 - 0.1log_2 0.1 = .461$. Also, the other bits are not compressed so effective compression is $(0.461 * 8 + 1 * 24)/32 = .867$. However, if we have common statistics for all bit positions, then the frequencies become 60% ones and 40% zeroes. With this frequency table it can be compressed to $-0.6log_2 0.6 - 0.4log_2 0.4 = .971$. Thus, we may compress to 86.7% instead of 97.1% of the total size. *Huffman coding* does slightly worse than this and this is the algorithm that should be used due to its greater speed of compression and expansion.

There are a couple of things to be noted about the format of the uncompressed data as well. It is not a good idea to store numbers in a pure binary format because redundancies in the bytes of lower significance will be lost. For example, the fact that most numbers are multiples of 5 may not be evident from seeing the lower order bytes when represented in binary. Thus, BCD (Binary coded decimal) appears to be a good format as it does not waste too much space and at the same time compressibility is not lost.

To provide the ordinal mapping, we will have to build an index to store for each block the row ordinal number of the starting value that is compressed in that block. The decompression may appear quite expensive. However, when clustering is done like that in GroupSet Indexes, there is a good chance that several close by values may actually contribute to the result and then the expense is not so much. This holds for all block level compression methods.

4.2.1 Another Redundancy

Values of some metric columns may have a strong relationship with the values of the dimensional tuples they join with. For example, the total sales at a particular shop will usually be of the same order. It cannot be a million one day and a hundred the next day. However, it may be quite different from other shops which may be smaller or bigger shops. Such a column is illustrated in Table 3. To compress such a BDI, we cluster these similar values by grouping by the dimensions on which the metric attribute depends. Now, the values of this column, within a particular group will vary only in the lesser significant bits. To exploit the large number of similar values in the higher order bits, we store the difference of each value with the average value of all the values in that group. Now, most higher order bits are turned to zeroes. Thus, we can use the same strategy as above of Huffman coding using a different frequency table for each byte. The average value can be stored in the index which maps ordinal numbers to disk locations.

4.3 The JDIs

In this section, we explore how the Join DataIndex may be compressed. Suppose no organization is done. Also assume that the cardinality of the referenced BDI is almost the same as 2^m where m is the width of the JDI column in bits. Then the JDI is a string from the m bit alphabet. No redundancy can be expected. It does not make sense for one to use too many bits for a JDI, though some bits may be kept extra for future growth of the warehouse. If for some reason this is the case then we can use the Huffman coding technique mentioned above for metric columns.

Thus, we must do some kind of organization of data. Consider a warehouse having a fact table F and 4 dimensional tables A, B, C, D. The fact table will have 4 JDIs J_A, J_B, J_C, J_D on A, B, C and D respectively.

Original Column	In index	Differenced data
7240	7000	240
7582		582
6995		-5
20000	20000	0
22050		2050
21075		1075
4095	4000	95
4172		172
3947		-53

Table 3: Another Redundancy in Numeric Columns

Let us sort the data on JDIs in ascending order by J_A, J_B, J_C, J_D . Also, the dimensional tables must be sorted using the hierarchy in their attributes (as described in the Section 4.4) so that related values are close together and the differences between successive JDI entries after sorting is small. When this is done, at least the JDIs J_A, J_B look extremely compressible. They will have groups of the same number repeated several times. Table 4 illustrates this.

Run Length Encoding looks ideal for such data. Another important observation is that the repeats are in sorted order. In other words if we store the Run-lengths and the literal which is repeating separately, the literals will be in sorted groups. This sorted group can be converted to a bitmap if it is dense resulting in greater compression. This is illustrated below. An additional advantage is that the bitmap may be anded with the rowset in memory while doing SJL, and if the resultant bitmap is empty, one may not read the run-lengths. Another possible way is to take successive differences modulo the size of the corresponding dimension table between the literals and then Huffman code them. It should be clear that this results in great compression of J_A, J_B and compressing them more is of no interest.

Example :
Let the Domain of the JDI be 1 to 10
The Original JDI sequence : 4 4 4 4 4 4 5 5 7 7 7 7 9 9 9 9   2 2 2 2 2 4 4 4
The Run-Lengths in the compresses version : 6 2 4 4   5 3
The Bitmaps will be : 0001101010   0101000000

If we observe J_C carefully, we find that the differences between successive values is much smaller than the actual values and it is not difficult to visualize that this will be the case in a large warehouse. Assume that the size of the dimensional table C is 7. Then, if we take successive differences modulo 7 we get the column as shown in Table 5.

All the values are now small, and we can use Huffman coding as we did in Section 4.2 to compress the JDI. Note that the next JDI, J_D is simply a random string with no characteristics and we cannot compress it.

J_A	J_B	J_C	J_D
1	1	1	8
1	1	4	6
1	1	7	14
1	2	2	7
1	2	3	10
1	2	5	5
2	1	1	8
2	1	3	1
2	1	4	15
2	1	5	12
2	3	2	8
2	3	4	9
2	3	4	7

Table 4: JDIs After Organization

Differences
1
3
3
2
1
2
3
2
1
1
4
2
0

Table 5: JDI J_C After Successive Differences

There are two problems with these techniques. The first is that only the first two or three JDIs by which you sort get compressed. The best we can do is to first group by the lower cardinality dimensional tables so that the runs are of the maximum size. Another problem is the high overhead of organization. Whenever new records are added, the whole fact table must be merge sorted with the data being added. This would involve reading and rewriting the entire warehouse data every time you update. A compromise would be to first group by the Time dimension. Then, all new tuples will be inserted towards the end and we have to merge sort only a small part of the entire data. An additional benefit of this is that all the other references to the date dimension table also get clustered. An example of this is the order-date, ship-date and commit-date in the TPC-D database. This should be fairly obvious since two objects ordered on the same day will be shipped at nearly the same day (at least the difference will not be large). Such a relationship could be expected among all time JDIs. So, if all the date columns are stored in a single JDI, the other columns can actually be represented by simply storing the signed difference between themselves and the sorted column. Obviously, since the differenced values will be much smaller they can be compressed using Huffman coding as we did for the numerical columns of the fact table in Section 4.2. Also, if we intend to store them in different JDIs, then we can apply the technique of differencing with an average value described in Section 4.2.1 on a disk block basis.

4.4 Dimensional BDIs

The dimensional tables usually consist of text and have a hierarchy among their columns. The values in columns that are higher in the hierarchy are likely to occur several times in that column. Thus, one may consider dictionary based methods to be the best. However, we may order the dimensional table according to the hierarchy, just as we did in the case of horizontal partitioning and groupset indexes (i.e. a group by on attributes $A_1, A_2 \ldots A_k$ where A_1 is highest in the hierarchy while A_k is the lowest). This will result in all repetitions getting clustered. Now, we may apply RLE. A perfect example is the time dimension table having a hierarchy of date, month, year, etc. If long runs are not always expected but there is a good possibility, it is better to do block level LZ77 as it subsumes Run Length Encoding and does well otherwise, too.

Another major issue is the representation of such a compressible BDI in memory. By representing the BDI in lesser memory we will effectively be reducing memory requirements of SJL. Thus, we may be able to use SJL, where otherwise we may have to use SJS. We can store such a BDI in memory using the same idea as the JDI. We can store all the distinct values in an in memory table and then instead of storing each value in the BDI, store a pointers to the corresponding entry in the in memory table. This may save a lot of space if there are lot of repetitions and the width of the table is much more than that of a pointer. This clearly results in SJL having to perform an extra pointer dereference.

One may consider run length encoding the column of pointers stored too. But then, the ordinal mapping is destroyed and an index must be provided or a binary search must be done. It may be noted that this kind of organization has several benefits like that in Groupset Indexes and horizontal partitioning. Besides, the organization may not be done every time you load new data. Its just that the new data will not be compressed as efficiently until a holiday when you can reorganize the entire database.

4.5 Making the Decisions

As we may observe, all these compression schemes may be simultaneously applied in the same star schema. The kind of organization involved in all of them is similar. However, there is no algorithm as such that can work out exactly what organization is to be done. We have given the characteristics expected by the compression algorithm and when a particular algorithm does particularly well or particularly badly. The DBA must use his knowledge of the data, and the conditions stated in this paper under which each scheme performs best to decide his organization. Some of the factors that influence the decisions and our recommendations are enumerated below.

- Size of the Dimension Tables: If a dimension table has columns that are quite large and may not fit in memory along with some other columns, one must consider horizontal partitioning. This may enable us to use SJL where SJS would have been necessary otherwise. Also, we recommend sorting first by smaller dimension tables.

- Expected Entropy: If a column is expected to have a very high entropy with respect to the model used for compressing that column, one may be better off not compressing it at all. As an example of a column that is not worth compressing, a column of width 1 byte having values uniformly distributed in the range 0 to 255.

- Columns accessed together: If most queries in the workload access some columns together, it is a wise choice to store these in a single BDI. Further, if their values are related, as they are usually very close to each other or even equal, then we may just store one value and the difference between them. Now, the difference values may be compressed using Huffman coding as suggested in Section 4.2 as they will usually be small.

- Fixed/Variable length columns: Fixed length columns become variable length as a result of compression. As a result, we lose the efficiency associated with fixed length columns. Thus, the cost of compressing (in terms of query processing time) fixed length columns is more than that of compressing variable length columns.

- Cost of Disk Space: If compression is adding overhead, the user will compromise on query performance if disk is precious to him.

- Frequency of updates: If the frequency of updates is very high and reorganization cost has to be kept at a minimum, then we must sort first by the time dimension.

- Workload: One must consider the queries that are usually going to be processed, while making decisions regarding the organization. For instance, the sort order may be determined as the order of the group-bys in most queries.

These decisions are very important as any compression scheme applied on data that was not expected by the compression method will result in lesser compression (or even expansion) and hence inefficiency in query processing.

Table	Type	No. of Rows
Main	Fact Table	26685
D1	Dim. Table	5
D2	Dim. Table	5
D3	Dim. Table	8
D4	Dim. Table	3443
D5	Dim. Table	20086

Table 6: The Tables in the Warehouse Data

Column	Type	Table	Technique	Original Size	Compressed Size	CR
Name	Text	D4	LZW	35.0K	12.0K	65%

Table 7: Results for Dimension Table: D4

An important benefit due to the isolation of columns provided by the DataIndexes is that columns that are not compressed do not suffer from any kind of inefficiency due to the compression of other columns of the same table. If the tables were not vertically partitioned, compressing one column would have an adverse effect on even queries involving only the uncompressed columns of the table. This is mainly because the records become variable length.

4.6 Performance On Sample Data

We have implemented these techniques, and this section gives a summary of how various techniques performed on real data. The data has some of characteristics of an actual warehouse. The data is from a popular paint company. It has recipes of 26,685 shades in terms of amounts of some primary dyes. The central fact table has 5 JDIs. Table 6 gives a description of the tables.

D4 contains only one attribute which is the name of the shade. The largest dimension table, D5, has around 3 to 4 records for every referenced key from the fact table describing which are the basic dyes to be used and in what quantity. The quantity column in this table is the only real numeric data we have.

Main, D4, D5 are the only tables large enough to be compressed. Let us begin with D4. It has only one column, called 'Name' which consists of text. Note that the data is represented in variable length format (not fixed length by padding with blanks) before compressing it. Each attribute is terminated by a newline to denote its end. The result of compressing this using Block level LZW is in Table 7.

Table D5 has four columns- C1, C2, C3, C4. C1 is the key which is referenced and has values from 1 to 5815. The table has 3 to 4 rows with the same value of C1. These are differentiated by their C2 value. Thus C1, C2 forms a candidate key. C3 is a JDI referencing a table with 16 rows. C4 is a float column denoting quantity of the referenced item. C4 was pre-processed and converted into an integer by multiplying by 10 and then the value was stored in BCD with no separators between them. The table was sorted by C1, C2. It so happens that since this is data created by the company, the table is also sorted by C1, C3. Thus, we can compress C1, C2, C3 by successive differencing and Huffman Coding it. C4 was directly Huffman coded as most of the values were very low. C1 has runs of single values and

Column	Type	Table	Technique	Original Size	Compressed Size	CR
C1	JDI	D5	Diff/Huff	10.0K	2.5K	75%
C1	JDI	D5	Diff/LZW	10.0K	1.6K	84%
C2	JDI	D5	Diff/Huff	10.0K	2.5K	75%
C2	JDI	D5	Diff/LZW	10.0K	1.8K	82%
C3	JDI	D5	Diff/Huff	20.0K	11.0K	45%
C3	JDI	D5	Diff/LZW	20.0K	12.5K	38%
C4	Numeric	D5	Huffman	40.0K	22.0K	45%
C4	Numeric	D5	LZW	40.0K	26.0K	35%

Table 8: Results for Dimension Table: D5

Column	Type	Table	Technique	Original Size	Compressed Size	CR
J1	JDI	Main	Diff/Huff	13K	5.0K	61%
J1	JDI	Main	Diff/LZW	13K	5.0K	61%
J4	JDI	Main	Diff/Huff	53K	14.5K	73%
J4	JDI	Main	Diff/LZW	53K	5.5K	90%
J5	JDI	Main	Diff/Huff	53K	20.0K	62%
J5	JDI	Main	Diff/LZW	53K	17.5K	67%

Table 9: Results for Fact Table: Main

C2 has sequences such as $1, 2, \ldots$. In such a case, after differencing we will have long runs of 0's and 1's respectively, which is a perfect opportunity for dictionary based techniques. Thus, dictionary techniques after differencing do slightly better compressing to less than 2K. Otherwise, Huffman Coding does better for the other columns. The results are summarized in Table 8.

Now we consider compressing the fact table, 'Main'. It only has 5 JDIs-J1 to J5 referencing the 5 dimension tables respectively. The table is sorted by the JDIs on J2, J3, J4 in that order. Again, since the data is created by the company, it so happens that J5 is also almost sorted. In fact, J5 is equal to J4 in several rows. Due to the very low cardinality of D2 and D3, J2 and J3 will have very long runs. As a result they are compressed to almost nothing as a result of run length encoding. The same experiments were done on the other JDIs-J1, J4, J5 as for the case of D5 and the results are summarized in Table 9.

However, an important observation while compressing J4 and J5 was that it was mostly populated with long sequences of successive numbers like $1, 2, 3 \ldots$. And due to the same reason as above we expect dictionary techniques to perform better on them. Indeed, the table above shows that block level LZW compressed the same to much lesser. Thus, when patterns like runs and sequences are expected to occur in the JDI, LZW should be used after taking successive differences. Otherwise, one should use Huffman Coding.

Overall, the entire data got compressed to around 30% to 35% of its original size. This implies a compression ratio of 65% to 70%, which is quite impressive. As stated previously this data has been synthetically created and is a lot more ordered than one can expect a warehouse to be. However, a 50%

compression ratio would not be surprising for most warehouses.

5 Conclusion and Future Work

The contribution of this work is twofold. First, In Section 2 we proposed a horizontal partitioning scheme in order to reduce the memory requirements of SJL enabling us to avoid using SJS as much as possible. Secondly, we described several compression algorithms that could be used to compress data warehouses stored as DataIndexes and discussed when each of them should be used. It might have appeared then that partitioning finely is a good idea. However, the more partitioning that is done, the more compression suffers. This is because the groups get split across several partitions and have to be coded separately in each partition. Thus, we must partition only to the extent that we don't have to do SJS for most queries, so that we get the efficient joining of SJL along with good compression.

We found that representing data as DataIndexes is very efficient, both in terms of space as well as query processing. It was only because of the DataIndex representation that things like run length coding on a column are possible. Secondly, different types of columns, with different semantics, could be compressed independently using different techniques which work best for that particular type of data to occupy lesser space than any general technique would otherwise result in. Another feature is that compressing some columns does not distribute their overhead to the other columns. In other words, we can compress some columns which we don't need very often and get the same performance as before on queries involving columns that are not compressed. Also, a large amount of data displays similarities (in the sense of compressibility) down a column and therefore we expect compression ratios of most compression techniques (even techniques of the future) to improve substantially when used on vertically partitioned data.

In the recent past, both processing power and I/O speeds have increased rapidly. Today, we cannot ignore the compression overhead in comparison to saving of I/O. A lot of research has to be done to design fast compression and decompression algorithms which give good compression ratios even at block level compression. We were particularly impressed by the hardware implementation of Huffman coding. However, their schemes had several drawbacks in the sense that the tree seemed to be hard coded into the chip. Also, no one uses Huffman coding to compress all their data today! It would be an interesting venture to figure out what makes the other techniques difficult to implement efficiently in hardware (at speeds as fast as Data Transfer Speeds of disks). More importantly, techniques must be designed which are implementable in hardware at high speeds and are good at compressing data that Huffman coding is not. In this way, they could complement each other and result in a system which stores data in little space with absolutely no extra time overhead.

Before designing a compression technique, one needs to figure out what kind of redundancy to expect in the data, for no single compression method can do well at all kinds of data. We did not have the opportunity to analyze data in actual warehouses. A good area for future research is to determine what other redundancies typically occur in warehouses, so that algorithms can then be designed to exploit these properties.

References

[BW98] A. Buchmann and Ming-Chuan Wu. *Encoded Bitmap Indexing for Data Warehouses.* Proc. ICDE, Orlando, February 1998.

[CY98] Chee-Yong Chan, Yannis E. Ioannidis. *Bitmap Index Design and Evaluation.* Proc. of SIGMOD 1998.

[HD52] Huffman D. A. *A Method for Construction of Minimum Redundancy Codes.* Proc. Inst. Electr. Radio Engg. 40.9 September 1952.

[IW94] B. Iyer, D. Wilhite *Data Compression Support in Databases.* Proc. of 20th VLDB Conference, 1994.

[KA98] Anindya Datta, Bongki Moon, Krithi Ramamritham, Helen Thomas, Igor Viguier. *"Have Your Data and Index It Too": Efficient Storage and Indexing for Data Warehouses.* Technical Report, 1998.

[KP98] Anindya Datta, Debra VanderMeer, Krithi Ramamritham, Bongki Moon. *Applying Parallel Processing Techniques in Data Warehousing and OLAP.* Submitted for Publication, 1998.

[ND97] Prasad Deshpande, Jeffrey Naughton, Amit Shukla, K. Ramasamy. *Caching Multidimensional Queries Using Chunks.* Proc. of SIGMOD, 1998.

[OG95] Patrick O'Neil, Goetz Graefe. *Multi-Table Joins Through Bitmapped Join Indices.* SIGMOD Record, September 1995.

[PQ97] Patrick O'Neil, Dallan Quass. *Improved Query Performance with Variant Indexes.* Proc. of SIGMOD, 1997.

[RH95] Gautam Ray, Jayant Haritsa, S Seshadri. *Database Compression: A Performance Enhancemant Tool.* Proc. of COMAD, 1995.

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

[WB94] M. Burrows, D. J. Wheeler. *A Block Sorting Lossless Data Compression Algorithm.* SRC research report 124, gatekeeper.dec.com/pub/DEC/SRC/research-reports/SRC-124.ps.Z

[WC87] Ian Witten, Radford Neal, John Cleary. *Arithmetic Coding for Data Compression.* Communications of the ACM June 1987.

[We84] Terry A. Welch. *A Technique for High Performance Data Compression.* IEEE Computer 17.6 June 1984.