Volume 2, Issue 6, June 2012 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Different Compression Techniques and Their Execution In Database Systems To Improve Performance

Naresh Kumar, Dr. Kapil Kr. Bansal
Assistant Professor
Department of Information Technology ,
SRM University, NCR Campus

ABSTRACT :Compression in traditional database systems is known to improve performance significantly [1, 4], it reduces the size of the data and improves I/O performance by reducing seek times (the data are stored nearer to each other), reducing transfer times (there is less data to transfer), and increasing buffer hit rate (a larger fraction of the DBMS fits in buffer pool). For queries that are I/O limited, the CPU overhead of decompression is often compensated for by the I/O improvements. We revisit this literature on compression in the context of column-oriented database systems. Storing data in columns presents a number of opportunities for improved performance from compression algorithms when compared to row-oriented architectures. In a column-oriented database, compression schemes that encode multiple values at once are natural. In a row-oriented database, such schemes do not work as well because an attribute is stored as a part of an entire tuple, so combining the same attribute from different tuples together into one value would require some way to "mix" tuples. Compression techniques for row-stores often employ dictionary schemes where a dictionary is used to code wide values in the attribute domain into smaller codes. For example, a simple dictionary for a string-typed column of colors might map "blue" to 0, "yellow" to 1, "green" to 2, and so on [1, 2]. Sometimes these schemes employ prefix-coding based on symbol frequencies (e.g., Huffman encoding [46]) or express values as small differences from some frame of reference and remove leading nulls from them

Key Words database ,compression,column oriented database , query execution etc.

I. INTRODUCTION

Our work also introduces a novel architecture for passing compressed data between operators that minimizes operator code complexity while maximizing opportunities for direct operation on compressed data. Previous work [40, 67, 32] also stresses the importance of insulating the higher levels of the DBMS code from the details of the compression technique. In general, this is accomplished by decompressing the data before it reaches the operators (unless dictionary compression is used and the data can be processed directly). However, in some cases increased performance can be obtained in query processing if operators can operate directly on compressed data (beyond simple dictionary schemes) and our work is the first to propose a solution to profit from these potential optimizations while keeping the higher levels of the DBMS as insulated as possible.

In this paper we briefly describe the compression schemes .For each scheme, we first give a brief description of the traditional version of the scheme as previously used in row store systems (and cite papers that provide more detail when possible). We then describe how the algorithm is used in the context of column-oriented databases.

II. Null Suppression

There are many variations on the null compression technique (see [4,5] for some examples), but the fundamental idea is that consecutive zeros or blanks in the data are deleted and replaced with a description of how many there were and where they existed. Generally, this technique performs well on data sets where zeros or blanks appear frequently. We chose to implement a column oriented version of the scheme described in [5]. Specifically, we field sizes to be variable and encode the number of bytes needed to store each field in a field prefix. This allows us to omit leading nulls needed to pad the allow data to a fixed size. For example, for integer types, rather than using the full 4 bytes to store the integer, we encoded the exact number of bytes needed using two bits (1, 2, 3, or 4 bytes) and placed these two bits before the integer. To stay byte-aligned we combined

these bits with the bits for three other integers (to make a full byte's worth of length information) and used a table to decode this length quickly as in [5].

III. Dictionary Encoding

Dictionary compression schemes are perhaps the most prevalent compression schemes found in databases today. These schemes replace frequent patterns with smaller codes for them. One example of such a scheme is the color-mapping given in the introduction. Other examples can be found in [2,4].

We implemented a column-optimized version of dictionary encoding. All of the row-oriented dictionary schemescited above have the limitation that they can only map attribute values from a single tuple to dictionary entries. This is because row-stores fundamentally are incapable of mixing attributes from more than one tuple in a single entry if other attributes of the tuples are not also included in the same entry (by definition of "row-store" – this statement does not hold for PAX-like [2] techniques that columnize blocks). Our dictionary encoding algorithm first calculates the number of bits, X, needed to encode a single attribute of the column (which can be calculated directly from the number of unique values of the attribute). It then calculateshow many of these X-bit encoded values can fit in 1, 2, 3, or 4 bytes. For example, if an attribute has 32 values, it canbe encoded in 5 bits, so 1 of these values can fit in 1 byte, 3 in 2 bytes, 4 in 3 bytes, or 6 in 4 bytes. We choose oneof these four options using the algorithm described in the next paragraph. Suppose that the 3-value/2-byte option was chosen. In that case, a mapping is created between every possible set of 3 5-bit values and the original 3 values. For example, if the value 1 is encoded by the 5 bits: 00000; the value 25 is encoded by the 5 bits: 00001; and thevalue 31 is encoded by the 5 bits 00010; then the dictionary would have the entry (read entries right-to-left)

X000000000100010 -> 31 25 1

where the X indicates an unused "wasted" bit. The decoding algorithm for this example is then straight-forward: readin 2-bytes and lookup entry in dictionary to get 3 values back at once. Our decision to keep data byte-aligned mightbe considered surprising in light of recent work that has shown that bit-shifting in the processor is relatively cheap.However our experiments show that column stores are so I/O efficient that even a small amount of compressionis enough to make queries on that column become CPU-limited (Zukowski et. al observe a similar result [3]) sothe I/O savings one obtains by not wasting the extra space are not important. Thus, we have found that it is worthbyte-aligning dictionary entries to obtain even modest CPU savings.

Cache-Conscious Optimization

The decision as to whether values should be packed into 1, 2, 3, or 4 bytes is decided by requiring the dictionary tofit in the L2 cache. In the above example, we fit each entry into 2 bytes and the number of dictionary entries is 32^3=32768. Therefore the size of the dictionary is 393216 bytes which is less than half of the L2 cache on our machine (1MB). Note that for cache sizes on current architectures, the 1 or 2 byte options will be used exclusively.

Parsing Into Single Values

Another convenient feature of this scheme is that it degrades gracefully into a single-entry per attribute scheme which is useful for operating directly on compressed data. For example, instead of decoding a 16-bit entry in the above example into the 3 original values, one could instead apply 3 masks (and corresponding bit-shifts) to get the three single attribute dictionary values. For example:

(X000000000100010 & 0000000000011111) >> 0 = 00010
(X000000000100010 & 0000001111100000) >> 5 = 00001
(X000000000100010 & 0111110000000000) >> 10 = 00000

These dictionary values in many cases can be operated on directly and lazily decompressed at the top of the query-plan tree.We chose not to use an order preserving dictionary encoding scheme such as ALM or ZIL since these schemes typically have variable-length dictionary entries and we prefer the performance advantages of having fixed length dictionary entries.

IV. Run-length Encoding

Run-length encoding compresses runs of the same value in a column to a compact singular representation. Thus, it is well-suited for columns that are sorted or that have reasonable-sized runs of the same value. These runs are replaced with triples: (value, start position, run length) where each element of the triple is given a fixed number of bits. When used in row-oriented systems, RLE is only used for large string attributes that have many blanks or repeated characters. But RLE can be much more widely used in column-oriented systems where attributes are stored consecutively and runs of the same value are common (especially in columns that have few distinct values). As described in , the C-Store architecture results in a high percentage of columns being sorted (or secondarily sorted) and thus provides many opportunities for RLE-type encoding.

V. Bit-Vector Encoding

Bit-vector encoding is most useful when columns have a limited number of possible data values (such as states in the US, or flag columns). In this type of encoding, a bit-string is associated with each value with a '1' in the corresponding position if that value appeared at that position and a '0' otherwise. For example, the following data:

1 1 3 2 2 3 1

would be represented as three bit-strings:

bit-string for value 1: 1100001

bit-string for value 2: 0001100

bit-string for value 3: 0010010

Since an extended version of this scheme can be used to index row-stores (so-called bit-map indices [5]), there has been much work on further compressing these bit-maps and the implications of this further compression on

Properties	Iterator Access	Block Information
Is One Value()	getNext()	getSize()
isValueSorted()	As Array()	getStartValue()
isPost Config()		getEndPosition()

Table : Compressed Block API

query performance [2,4,5]; however, the most recent work in this area indicates thatone needs the bit-maps to be fairly sparse (on the order of 1 bit in 1000) in order for query performance to not behindered by this further compression, and since we only use this scheme when the column cardinality is low, our bit-maps are relatively dense and we choose not to perform further compression.

VI. Heavyweight Compression Schemes

Lempel-Ziv Encoding. Lempel-Ziv ([2,3]) compression is the most widely used technique for lossless file compression. This is the algorithm upon which the UNIX command gzip is based. Lempel-Ziv takes variable sized patterns and replaces them with fixed length codes. This is in contrast to Huffman encoding which produces variable sized codes. Lempel-Ziv encoding does not require knowledge about pattern frequencies in advance; it builds the pattern table dynamically as it encodes the data. The basic idea is to parse the input sequence into non-overlapping blocks of different lengths while constructing a dictionary of blocks seen thus far. Subsequent appearances of these blocks are replaced by a pointer to an earlier occurrence of the same block. We refer the reader to [3,4] for more details.For our experiments, we used a freely available version of the Lempel-Ziv algorithm [3] that is optimized for decompression performance (we found it to be much faster than UNIX gzip).

We experimented with several other heavyweight compression schemes, including Huffman and Arithmetic encoding, but found that their decompression costs were prohibitively expensive for use inside of a database system.

VII. EXPERMENTED RESULTS

In this experiment, we ran a simple aggregation on a single column of data encoded with each of the six encoding schemes described earlier. We ran on generated data and required that the column be decompressed as it was brought off disk. The query that we ran was simply:

```
SELECT SUM(C)
FROM TABLE
GROUP BY C
```

The column that we are aggregating has 100 million 32-bit integer values. Since most columns in C-Store projections have some kind of order , we assume sorted runs of size X (we vary X). For example, if column C is tertiarily sorted and the first column in the projection has 500 unique values and the second column in the projection has 1000 unique values then C will have average sorted runs of size 100000000/(500*1000)=200. If C itself has 10
unique values, then within each of these sorted runs, each value would appear 20 times. Since bit-vector compression is only designed to be able to run on columns with few distinct values, in our first set of experiments, we allowed the number of distinct values in C to vary between 2 and 40 (so that we could directly compare all the introduced compression techniques). Also, in most data-warehousing environments, there are a large number of columns with few distinct values; for example, in the TPC-H lineitem fact table, 25% of the columns have fewer than 50 distinct values. We experiment with columns with a higher number of distinct values in .

We experimented with four sorted run lengths in C: 50, 100, 500, and 1000. We compressed the data in each of the following six ways: Null suppression, Lempel-Ziv, RLE, bit-vector, dictionary, and no compression. The sizes of the compressed columns can be shown in Figures

for different cardinalities of C (here, we use *cardinality* to mean the number of distinct values). We omit the plots for the 100 and 500 sorted runs cases as they follow the trends observed in. In these experiments, dictionary and LZ compression consistently get the highest compression ratios. Dictionary does a slightly better job compressing the data than the heavy-weight LZ scheme at low column cardinalities since our implementation of LZ will occasionally leave some empty space at the
end of a page if it gets compressed more than surrounding pages

VIII. CONCLUSIONS

In summary, this paper shows that significant database performance gains can be had by implementing light-weight compression schemes and operators that work directly on compressed data. By classifying compression schemes according to a set of basic properties, we were able to extend C-Store to perform this direct operation without requiring new operator code for each compression scheme. Furthermore, our focus on column-oriented compression allowed us to demonstrate that the performance benefits of operating directly on compressed data in column-oriented schemes is much greater than the benefit in operating directly on row-oriented schemes.

Hence, we see this work as an important step in understanding the substantial performance benefits of columnoriented database designs. Although this paper focused on fairly simple queries so as to carefully distill the performance characteristics of column-oriented compression.

REFERENCES:

1. D. Huffman. A method for the construction of minimum-redundancy codes. Proc. IRE, 40(9):1098-1101, September 1952.

2 Balakrishna R. Iyer and David Wilhite. Data compression support in databases. In *VLDB '94*, pages 695–704, 1994.

3 Theodore Johnson. Performance measurements of compressed bitmap indices. In *VLDB*, pages 278–289, 1999.

4 Setrag Khoshafian, George Copeland, Thomas Jagodis, Haran Boral, and Patrick Valduriez. A query processing strategy for the decomposed storage model. In *ICDE*, pages 636–643, 1987.

5 Roger MacNicol and Blaine French. Sybase IQ multiplex - designed for analytics. In *VLDB, pp. 1227-1230,* 2004.

6 A. Moffat and J. Zobel. Compression and fast indexing for multi-gigabyte text databases. *Australian*

