Database compression techniques for performance optimization

Sushila Aghav
Faculty, Computer, MITcollege Of Engineering, MITCOE
Pune, Maharashtra India
sushila_aghav@yahoo.com

Abstract— **Data stored in databases keep growing as a result of businesses requirements for more information. A big portion of the cost of keeping large amounts of data is in the cost of disk systems, and the resources utilized in managing that data.**

This paper introduces various compression techniques for data stored in row oriented as well as column-oriented databases.

Keeping data in this compressed format as it is operated upon has been shown to improve query performance by up to an order of magnitude. Intuitively, data stored in columns is more Compressible than data stored in rows. Compression algorithms perform better on data with low information entropy (high data value locality) i.e are used for optimization purpose

Keywords—Column Stores, row-stores, compression, decompression, Cache-Conscious Optimisation

I. INTRODUCTION

Commercially available relational database systems have not heavily utilized compression techniques on data stored in relational tables. One reason is that the trade-off between time and space for compression is not always attractive for relational databases. A typical compression technique may offer space savings, but only at a cost of much increased query time against the data. Furthermore, many of the standard techniques do not even guarantee that data size does not increase after compression.

This paper introduces different compression techniques for different domains of databases.

II. HISTORY OF DATABASE COMPRESSION

Data compression algorithms (such as the Huffman algorithm) , have been around for nearly a century, but only today are they being put to use within mainstream information systems processing. The entire industrial strength database offers some for of data compression (Oracle, DB2, CA-IDMS), while they are unknown within simple data engines such as Microsoft Access and SQL Server[7]. There are several places where data can be compressed, either external to the database, or internally, within the DBMS software:

A. *Physical database compression*

- Hardware assisted compression - IMS, the first commercially available database offers Hardware Assisted Data Compression (HDC), which interfaces with the 3380 DASD to compress IMS blocks at the hardware level, completely transparent to the database engine.
- Block/page level compression –This historical database compression [7][8] uses external mechanisms that are invisible to the database. As blocks are written from the database, user exits invoke compression routines to store the compressed block on disk. Examples include CLEMCOMP, Press pack and InfoPak. In 1993, the popular DB2 offers built-in table space level compression using a COMPRESS DDL keyword.

B. *Logical database compression*

Internal Database compression operates within the DBMS software, and write pre-compressed block directly to DASD [7]:

- Table/segment-level compression - A database administrator has always had the ability to remove excess empty space with table blocks by using Data Pump reorganization utility (dbms_redefinition). By adjusting storage parameters, the DBA can tightly pack rows onto data blocks. For example, using Oracle9i table-level compression utility using a table DDL COMPRESS keyword
- Row level compression - In 2006, DB2 extended their page-level compression with row-level compression. Oracle offers a row-level compression routine.

III. COMPRESSION SCHEMES

In this section we briefly describe the compression schemes Column oriented databases. For each scheme, we first give a brief description of the traditional version of the scheme as previously used in row store systems [1].

A. *Null Suppression*

There are many variations on the null compression technique, 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. Specifically, we allow 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 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, or4 bytes) and placed these two bits before the integer. To stay byte-aligned, combine 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.

B. Dictionary Encoding

Dictionary compression schemes[1] are perhaps the most prevalent compression schemes found in databases today. These schemes replace frequent patterns with smaller codes for them.

A column-optimized version of dictionary encoding is new Implementation. All of the row-oriented dictionary schemes 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 techniques that columnize blocks).

1) *Dictionary encoding algorithm*

Dictionary encoding algorithm [3] 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 calculates how 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 can be 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. 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 the value 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 straightforward: read in 2-bytes and lookup entry in dictionary to get 3 values back at once.

Column stores are so I/O efficient that even a small amount of compression is enough to make queries on that column become CPU-limited so the I/O savings one obtains by not wasting the extra space are not important. Thus, it is worth byte-aligning dictionary entries to obtain even modest CPU savings.

2) *Cache-Conscious Optimization*[1][2]

The decision as to whether values should be packed into 1, 2, 3, or 4 bytes is decided by requiring the dictionary to fit in the L2 cache. In the above example, we fit each entry into 2 bytes and the number of dictionary entries is 323 = 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.

3) *Parsing Into Single Values*[1]

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.

C. Run-length Encoding

Run-length encoding compresses [1] 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).

D. Bit-Vector Encoding

Bit-vector encoding[2][4] 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), there has been much work on further compressing these bit-maps and the implications of this further compression on query performance however, the most recent work in this area indicates that one needs the bit-maps to be fairly sparse (on the order of 1 bit in 1000) in order for query performance to not be hindered by this further compression, and since we **only use this scheme when the column cardinality is low, bit-maps are relatively dense and when not to perform further compression.**

E. Heavyweight Compression Schemes

1) *Lempel-Ziv Encoding*[9][10]

Compression is the most widely used technique for loss less 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. A pointer replaces subsequent appearances of these blocks to an earlier occurrence of the same block.

F. *Hybrid columnar compression*

Traditionally, database table rows have been stored in blocks, as shown in Figure 1. Typically, a row is fully contained in a block, with the columns of the row stored next to each other[7]. However, when the row becomes too large to fit into a block, the row overflows into the next block—a phenomenon known as row chaining—but the organization of the columns being stored next to each other still remains the same. This compression mechanism replaces a value in a row with a much smaller symbol, reducing the length of the row. In real-world situations, however, data is more often repeated in columns, not rows. For example, here is the data for a simple (abbreviated) table:

```
FIRST_NAME     LAST_NAME
----------     ---------
Albert         Smith
Bernie         Smith
Charles        Smith
David          Smith
John           Smith
... and so on ...
```

The Smith value repeats many times, so a great deal of compression can be achieved by replacing the Smith value with a much smaller symbol. And because the same symbol can represent all of the repeated Smith values, fewer unique symbols will need to be stored, reducing the size of the compressed data significantly.

With Hybrid Columnar Compression, a column vector is created for each column, compresses the column vectors, and stores the column vectors in data blocks. The collection of blocks is called a compression unit. The blocks in a compression unit contain all the columns for a set of rows, as shown in Figure 1. (In Hybrid Columnar Compression, a row typically spans several data blocks.)

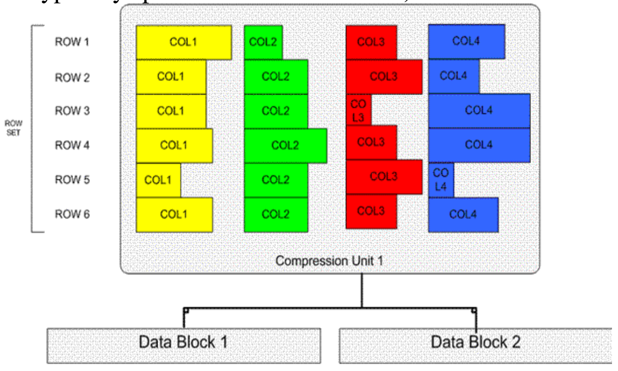


Figure 1. Compression Unit in hybrid columnar compression

1) *Types of Hybrid Columnar Compression*

Hybrid Columnar Compression [5][6] comes in two basic flavors: warehouse compression and archive compression.

a) Warehouse compression: For warehouse compression, the compression algorithm has been optimized for query performance, specifically for scan-oriented queries used heavily in data warehouses. **This approach is ideal for tables that will be queried frequently.**

Here is how you create a table with warehouse compression:

```
CREATE TABLE XXX
COMPRESS FOR QUERY
AS
SELECT * FROM YYY;
```

b) Archive compression: With archive compression, the compression algorithm has been optimized for maximum storage savings. **This approach is ideal for tables that are infrequently accessed. (Note that for compressing or decompressing data, archive compression may consume a significant amount of CPU compared to warehouse compression.)**

Here is how you create a table with archive compression:

```
CREATE TABLE XXX
COMPRESS FOR ARCHIVE
AS
SELECT * FROM YYY;
```

When a table compressed with Hybrid Columnar Compression is read, the CPU consumption may be higher than for an uncompressed table. However, because the number of blocks returned by a query against a compressed table is significantly lower, the logical reads and consistent gets are lower as well, often resulting in a reduction in both CPU consumption and I/O. So, the overall CPU consumption may actually be lower for queries against tables compressed with Hybrid Columnar Compression.

G. *OLTP Table Compression*

OLTP Table Compression [6] uses a unique compression algorithm specifically designed to work with OLTP applications. The algorithm works by eliminating duplicate values within a database block, even across multiple columns. Compressed blocks contain a structure called a symbol table that maintains compression metadata. When a block is compressed, duplicate values are eliminated by first adding a single copy of the duplicate value to the symbol table. Each duplicate value is then replaced by a short reference to the appropriate entry in the symbol table. Through this innovative design, compressed data is self-contained within the database block as the metadata used to translate compressed data into its original state is stored in the block. When compared with competing compression algorithms that maintain a global database symbol table. This approach offers significant performance benefits by not introducing additional I/O when accessing compressed data.

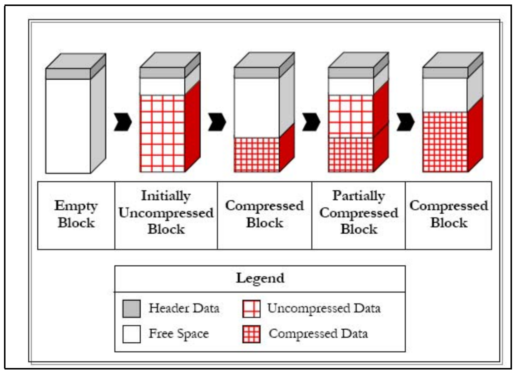
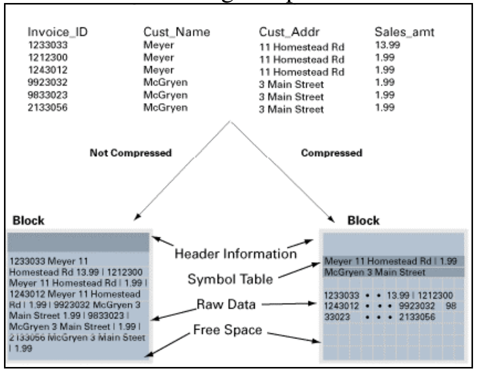


Figure 2. OLTP Table Compression Process

Here is how you create a table with archive compression OLTP Table Compression Syntax

```
CREATE TABLE emp (
emp_id NUMBER
, first_name VARCHAR2(128)
, last_name VARCHAR2(128)
) COMPRESS FOR OLTP;
```

When a block is compressed, duplicate values are eliminated by first adding a single copy of the duplicate value to the symbol table. Each duplicate value is then replaced by a short reference to the appropriate entry in the symbol table as shown in fig.3. Through this innovative design, compressed data is self-contained within the database block as the metadata used to translate compressed data into its original state is stored in the block. When compared with competing compression algorithms that maintain a global database symbol table. This approach offers significant performance benefits by not introducing additional I/O when accessing compressed data.



Invoice_ID	Cust_Name	Cust_Addr	Sales_amt
1233033	Meyer	11 Homestead Rd	13.99
1212300	Meyer	11 Homestead Rd	1.99
1243012	Meyer	11 Homestead Rd	1.99
9923032	McGryen	3 Main Street	1.99
9833023	McGryen	3 Main Street	1.99
2133056	McGryen	3 Main Street	1.99

Not Compressed
Compressed
Block
Block
Header Information
Symbol Table
Raw Data
Free Space
1233033 Meyer 11 Homestead Rd 13.99 | 1212300 Meyer 11 Homestead Rd | 1.99 | 1243012 Meyer 11 Homestead Rd | 1.99 | 9923032 McGryen 3 Main Street 1.99 | 9833023 | McGryen 3 Main Street | 1.99 | 2133056 McGryen 3 Main Steet | 1.99
Meyer 11 Homestead Rd | 1.99
McGryen 3 Main Street
1233033 • • 13.99 | 1212300
1243012 • • • 9923032 98
33023 • • • 2133056

Figure 3. Non-compressed Block Vs OLTP Compressed Block

IV. Conclusions

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. In addition to these traditional techniques, column-stores are also well-suited to compression schemes that compress values from more than one row at a time. This allows for a larger variety of viable compression algorithms.

Run-length encoding (RLE), where repeats of the same element are expressed as (value, run-length) pairs, is an attractive approach for compressing sorted data in a column-store. Similarly, improvements to traditional compression algorithms that allow basic *symbols* to span more than one column entry are also possible in a column-store.

Further, the CPU overhead of iterating through a page of column values tends to be less than that of iterating through a page of tuples (especially when all values in a column are the same size), allowing for increased decompression speed by using vectorized code

Compression schemes also improve CPU performance by allowing database operators to operate directly on compressed data.

This paper shows that implementing lightweight compression schemes and operators that work directly on compressed data can have significant database performance gains.

References

[1] D. J. Abadi. Query execution in column-oriented database systems. MIT PhD Dissertation, 2008. PhD Thesis.

[2] D. J. Abadi, S. R. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In SIGMOD, pages 671–682, 2006.

[3] M. Zukowski, P. A. Boncz, N. Nes, and S. Heman. MonetDB/X100 - A DBMS In The CPU Cache. IEEE Data Engineering Bulletin, 28(2):17–22, June 2005.

[4] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented DBMS. In ICDE, pages 466–475, 2007.

[5] Oracle Corporation. Oracle 9i Database for Data Warehousing and Business Intelligence. White Paper. http://www.oracle.com/solutions/business_intelligence/Oracle9idw_bwp.

[6] Oracle White Paper – Advanced Compression with Oracle Database 11g Release 2

[7] Oracle 11g Data Compression Tips for the Database Administrator Oracle 11g Tips by Burleson Consulting

[8] Query Execution in Column-Oriented Database Systems by Daniel J. Abadi

[9] A universal algorithm for sequential data compression. Jacob Ziv and Abraham Lempel IEEE Transactions on Information Theory, 23(3):337–343, 1977.

[10] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory, 24(5):530–536, 1978.