Volume 4, No. 7, July 2013
Journal of Global Research in Computer Science
ISSN-2229-371X
REVIEW ARTICLE
Available Online at www.jgrcs.info

A SHORT SURVEY OF DATA COMPRESSION TECHNIQUES FOR COLUMN ORIENTED DATABASES

Priyanka Raichand* and Rinkle Rani Aggarwal
Department of Computer Science and Engineering
Thapar University, Patiala
raichand.priyanka@gmail.com, raggarwal@thapar.edu

Abstract: Column oriented data-stores has all values of a single column are stored as a row followed by all values of the next column. Such way of storing records helps in data compression since values of the same column are of the similar type and may repeat. This Paper surveys the various data compression techniques in column oriented databases. Data compression is efficiently used to save storage space and network bandwidth. It improves the performance of query execution. Specialized algorithms based on the type of the data stored in column results in immense improvements in compression ratios. Higher-up ratios result in more efficient bandwidth usage.

Keywords: Compression, Column oriented databases, Decompression, NoSQL, HBase

INTRODUCTION

Data size in data-stores have been ever increasing due to increased networking and business demands and thus increasing cost of resources needed in terms of space and network utilization. To store terabytes of data, especially of the type human-readable text, it is beneficial to compress the data to gain significant savings in required raw storage. Compression techniques have not been considerably used in traditional relational database systems. The exchange between time and space for compression is not much pleasing for relational databases. Whereas storing data in columns introduce a number of possibilities for better performance from compression algorithms. Column-oriented databases save the data by grouping in columns. Later column values are stored consecutively on disk in contrast to row-oriented approach of databases which store entire rows contiguously [1].

HBase is one of the most prominent NoSQL column oriented data- store. In this paper, we have introduced compression techniques in the context of column oriented database systems. In the next section, we briefly layout related work. In Section 3, we identify various data compressing techniques well suited for column oriented data-stores. Section 4 briefly indicates the Compression schemes that can be used in HBase. We offer our conclusions in Section 5.

RELATED WORK

Research in database compression has been approximately for nearly a century though compression techniques were not frequently used until 1990's [2]. It's only today that these are being put to use inside important information systems. It may be so because earlier most of the was focused on cutting down the size of the stored data, but in 90's researchers started to concentrate on affects of compression on databases performance [10, 11, 13, 12]. Few researchers have investigated the effect of compression on database systems and their performance [14, 15, 13]. These Research work have discovered that compression does reduce I/O cost but if the cost of for compressing/decompressing outbalance this saved cost, then it results in reduced overall performance of the database. This trade-off becomes more favourable for compression with the improvements in CPU speed [20]. Many researchers have chew over text compression schemes such as Huffman encoding that is based on letter frequency. String matching schemes are being looked by more recent research in compression [16, 17, 18, 19].

Along with these traditional techniques, column oriented databases are also substantially best for compression schemes that compress data from more than one row at a time thus allowing many great kind of workable compression algorithms. For example, for compressing sorted data in a column oriented database, run-length encoding (RLE), where repeats of the same element are expressed as (value, run-length) pairs, is a pleasing technique [2]. We can see generally higher compression ratios in column oriented datastores because consecutive values of the same column are of the similar type and may repeat, whereas adjacent attributes in a tuple are not [21]. The overhead incurred by CPU for iterating through a page of column values (particularly when all column values are the same size) tends to be less. Column oriented databases can store different columns in unlike sort-orders [22], further enhancing the possibility for compression. Column oriented compression techniques also improve CPU performance by allowing operations directly on compressed data. This is particularly true for compression schemes like run length encoding that refer to multiple entries with the same value in a single record [2].

COMPRESSION SCHEMES FOR COLUMN ORIENTED DATABASES

This section concisely describes various compression

schemes for Column oriented databases. For each scheme, we firstly present a brief description of the traditional version of the scheme as previously used in row oriented database systems and later in regard with column oriented databases [4].

Dictionary Encoding

Today's database systems perhaps Dictionary encoding schemes are most dominant types of compression schemes. These schemes replace frequent patterns with smaller codes. A column-optimized variant of dictionary encoding is new Implementation. Row oriented datastores are basically not capable of blending attributes from more than one tuple in a single entry thus making dictionary encoding schemes not to function fully as they can only map attribute values from a single tuple to dictionary entries.

1) *Dictionary Encoding Algorithm:* Dictionary encoding algorithm [4] 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,**[**4] 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 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 oriented databases are quite I/O efficient that queries on the column become CPU limited after applying even small amount of compression[6]. So the I/O savings that one get by not wasting the extra space is unimportant. Thus, it is worth byte-aligning dictionary entries to obtain even modest CPU savings [2].

2) *Cache-Conscious Optimization [4][5]:* 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.

Run Length Encoding

Run Length Encoding (RLE) is a simple and popular data compression algorithm. Run-length encoding compresses [2] based on the idea of replacing the same long sequence 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. In row-oriented systems, RLE is only used for large string attributes that have many blanks or repeated characters. But in column oriented RLE can be much more greatly used systems where attributes are stored consecutively and runs of the same value are common (mainly in columns that have less distinct values).

Null Suppression

Null compression scheme has many variants but the basic logic is to replace consecutive zeros or blanks in the data are deleted and replaced with a description of how many there were and where they existed **[**2**].** Naturally, this technique works great on data sets where zeros or blanks appear frequently. Variable field sizes are encoded in the number of bytes needed to store each field in a field prefix. This allows us to exclude heading nulls needed to pad the data to a fixed size. For example, [2] 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 as prefix of the integer.

Lempel-Ziv Encoding

The Lempel-Ziv compression algorithms were developed in 1977-78. Lempel-Ziv [7, 8] compression is the most widely used technique for lossless file compression. The UNIX command gzip is based upon this algorithm only. Lempel-Ziv replaces variable sized patterns with fixed length codes unlike to Huffman encoding which produces variable sized codes. In Lempel-Ziv encoding knowledge about pattern frequencies in advance is not an requirement as it builds the pattern table dynamically as it encodes the data. The main idea is to analyse the input sequence into non-overlapping blocks of different lengths and constructing a dictionary of blocks seen thus far. Later on occurrences of these blocks are replaced by a pointer to an earlier occurrence of the same block.

Hybrid Columnar Compression

Generally, database table rows are stored in blocks. Typically, a row is fully contained in a block, and the columns of the row stored next to each other [9]. But too large rows can't fit in a block so results in spanning of the row to next block known as row chaining—but there is no change in the organization of the columns being stored next to each other. This compression mechanism replaces a value in a row with a much smaller symbol, thus reducing the length of the row. A great deal of compression can be achieved by replacing the repeated value with a much smaller symbol. In Hybrid Columnar Compression, we get a column vector for each column, compress the column vectors, and store the column vectors in data blocks. This collection of data blocks is known a compression unit. The blocks in a compression unit contain all the columns for a set of rows as shown in the Fig 1.

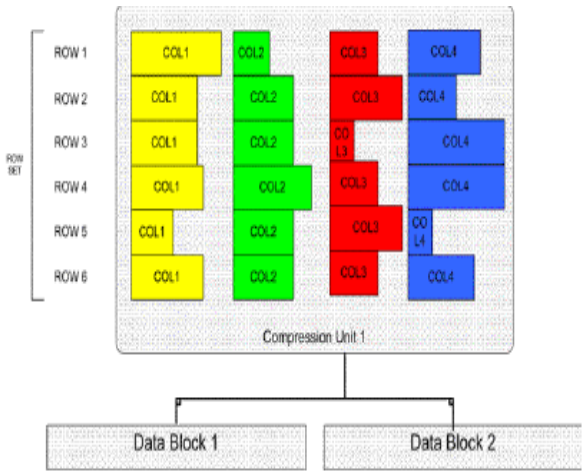

Fig 1. Compression Unit in hybrid columnar compression

COMPRESSION SCHEMES IN HBASE

HBase is the Hadoop database. It is a distributed, scalable Big Data store. We can use HBase in random, real-time read/write access to our Big Data. An open-source follow up of BigTable, HBase uses a data model very similar to that of BigTable. A data row in HBase is a sortable row key and a variable number of columns, which are further grouped into sets called column families. Each data cell in HBase can contain multiple versions of the same data which are indexed by timestamps. Fig. 2(a) represents an HBase table here as a sorted map in contrast to relational databases whose data is usually represented as a table form. Data cell in the table can be viewed as a key value pair where the key is a combination of row key, column and timestamp; and the value is an uninterpreted array of bytes Fig. 2(b).

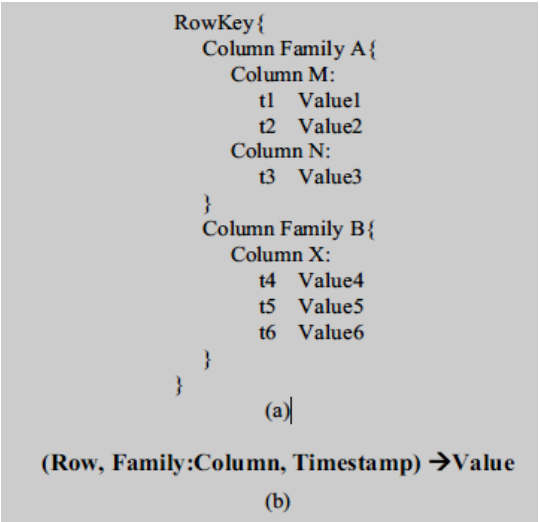

Fig 2. HBase data model [3]

One of the most important features of HBase is the use of data compression. It's important because [1]:
1. Compression reduces the number of bytes written to/read from HDFS.
2. Saves disk usage.
3. Improves the efficiency of network bandwidth when getting data from a remote server.
HBase comes with support for a number of compression algorithms that can be enabled at the column family level.

1) *Available Codec :* HBase supports the GZip and LZO, Snappy codec. Before moving ahead on the details of these codecs lets to see the compression algorithm comparison Google published in 2005

Table1. Comparison of compression algorithms

Algorithm	% remaining	Encoding	Decoding
GZIP	13.4%	21 MB/s	118 MB/s
LZO	20.5%	135 MB/s	410 MB/s
Snappy	22.2%	172 MB/s	409 MB/s

We can see from Table 1 that some of the algorithms have a better compression ratio while others are faster during encoding and a lot faster during decoding [1].

Snappy: With Snappy, released by Google under the BSD License, we got access to the same compression used by Bigtable (*Zippy*). It behaves perfectly to provide high speeds and reasonable compression. The code is written in C++.

LZO: Lempel-Ziv-Oberhumer (LZO) is a lossless data compression algorithm. It is focused on decompression speed, and written in ANSI C. HBase is not shiped with LZO because of licensing issues: HBase uses the Apache License, while LZO is using the incompatible GNU General Public License (GPL). By adding LZO compression support, HBase StoreFiles (Hfiles) uses LZO compression on blocks as they are written. HBase uses the native LZO library to perform the compression, while the native library is loaded by HBase via the hadoop-lzo Java library that we built [1**]**.

GZIP: The GZIP compression algorithm compresses better than Snappy or LZO, but is slower in comparison. It comes with an additional savings in storage space.

CONCLUSIONS

In this paper, we have outlined few fairly simple techniques to achieve database performance improvements by data compression. These techniques not only reduce space requirements on disk and I/O performance when measured in records per time for permanent and temporary data, they also reduce requirements of memory, thus reducing the number of buffer faults resulting in I/O. column oriented are well suited to compression schemes that compress values from more than one row at a time. Compression schemes also improve CPU performance by allowing database operators to operate directly on compressed data. Various compression codecs are available to be used with HBase, including LZO, Snappy and GZIP. In HBase Compression codecs work best if they

can decide how much data is enough to achieve an efficient compression ratio. HFiles can be compressed and stored on HDFS. This helps by saving on disk I/O and instead paying with a higher CPU utilization for compression and decompression while writing/reading data. LZO and Snappy have comparable compression ratios and encoding/decoding speeds.

REFERENCES

[1] (Book) "HBase: The Definitive Guide" (2nd edition). Lars George. O'Reilly Media, Inc., 2011.

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

[3] Chongxin Li. "Transforming Relational database into HBase- A Case Study".

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

[5] 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.

[6] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In ICDE, 2006.

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

[8] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. IEEE Transactionson Information Theory, 24(5):530–536, 1978.

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

[10] G.Graefe and L.Shapiro. Data compression and database performance. In ACM/IEEE-CS Symp. On Applied computing pages 22 -27, April 1991.

[11] B. R. Iyer and D. Wilhite. Data compression support in databases. In VLDB '94, pages 695–704, 1994.

[12] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In ICDE '98, pages 370–379, 1998.

[13] G. Ray, J. R. Haritsa, and S. Seshadri. Database compression: A performance enhancement tool. In COMAD, 1995.

[14] C. A. Lynch and E. B. Brownrigg. Application of data compression to a large bibliographic data base. In VLDB '81, Cannes, France, pages 435–447, 1981.

[15] P. O'Neil and D. Quass. Improved query performance with variant indexes. In SIGMOD, pages 38–49, 1997.

[16] A. Moffat and J. Zobel. Compression and fast indexing for multi-gigabyte text databases. Australian Computer Journal, 26(1):1–9, 1994.

[17] K. Wu, E. Otoo, and A. Shoshani. Compressed bitmap indices for efficient query processing. Technical Report LBNL-47807, 2001.

[18] K. Wu, E. Otoo, A. Shoshani, and H. Nordberg. Notes on design and implementation of compressed bit vectors. Technical Report LBNL/PUB-3161, 2001.

[19] A. Zandi, B. R. Iyer, and G. G. Langdon Jr. Sort order preserving data compression for extended alphabets. In Data Compression Conference, pages 330–339, 1993.

[20] P. Boncz, S. Manegold, and M. Kersten. Database architecture optimized for the new bottleneck: Memory access. In VLDB, pages 54–65, 1999.

[21] R. MacNicol and B. French. Sybase IQ multiplex - designed for analytics. In VLDB, pages 1227–1230, 2004.

[22] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A column-oriented DBMS. In VLDB, pages 553–564, 2005.