Fast Retrieval with Column Store using RLE Compression Algorithm

Ishtiaq Ahmed
Integral University
Dasauli, Kursi Road
Lucknow (UP) India

Sheesh Ahmad, Ph.D
Integral University
Dasauli, Kursi Road
Lucknow (UP) India

Durga Shankar Shukla
Integral University
Dasauli, Kursi Road
Lucknow (UP) India

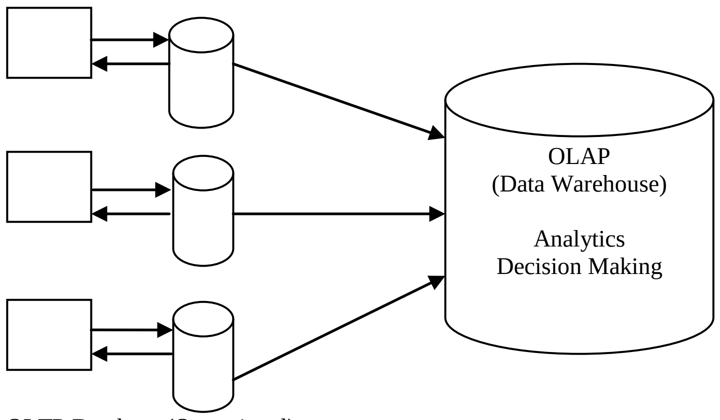
ABSTRACT
Column oriented database have continued to grow over the past few decades. C-Store, Vertica Monet DB and Lucid DB are popular open source column oriented database. Column-store in a nutshell, store each attribute values belonging to same column contiguously. Since column data is uniform type therefore, there are some opportunities for storage size optimization in Column-store, many renowned compression schemes such as RLE & LZW that make use of similarity of adjacent data to compress. Good Compression can also be achieved using bitmap index by order of magnitude through the sorting. The Run Length Encoding works best for the columns of ordered data, or data with few distinct values. This ensures long runs of identical values which RLE compresses quite well. In this paper we have put an effort to build a simulation of Column-Store and applied the best bitmap compression technique RLE which further improves the retrieval time.

General Terms
Your general terms must be any term which can be used for general classification of the submitted material such as Pattern Recognition, Security, Algorithms et. al.

Keywords
Bitmap, Column-Store, LZW, OLAP, OLTP, RLE

1. INTRODUCTION
Column –Oriented database has drawn a lot of attention in last few years. The source of column-oriented database systems can be seen beginning from 1970s, but it was not until 2000s that some researches and applications started to be done. In the past recent years some column store databases namely MonetDB [1] [2] and C-Store [3] has been introduced by their authors, with the claim that their performance gains are quite noticeable against traditional approaches. The column oriented database specifically designed for analytic purpose overcome the flaws encountered in traditional DBMS by storing, managing, querying, data based on column instead of row. Column-Stores approach, store each column separately rather than storing entire row i.e. instead of retrieving a record or row at a time, an entire column is retrieved, only necessary columns in a query are accessed rather entire rows, I/O activities as well as overall query response time is reduced and access becomes faster because much more relevant Column can be accessed in a shorter period of time [4]. Moreover there are some opportunities for storage size optimizations available in column-store because data in a column oriented database can be better compressed than those in a row-oriented database, values in a column are much more homogenous than in a row. The compression of a column-oriented database may reduce its size up to 20 times, this thing providing a higher performance and reduced storage costs [5][6].

Column oriented architecture is more suitable for data warehousing with selective access to small number of attributes. While row-oriented is better solution for OLTP systems, In such architecture all attributes are written on a disk in single command that requiring high performance for writing operations. For OLAP system, designed for analytical purposes, which involve processing of large number of values of few columns, a column –oriented is better solution .Indeed Column-Oriented has enabled highly complex query environments that support strategic and operational decisions can be used to achieve a competitive edge by better understanding customers, competition, risk positions, revenue leaks, and fraud. Column oriented DBMS allow to perform these data analytics [6] [7].



OLTP Database (Operational)

Fig 1: OLTP & OLAP Database Overview

The internal structure of Column-Store will be better understood by simulating column stores-inside row store. Star Schema Benchmark [6] is recently proposed data-warehousing benchmark that has been implemented with column-Oriented internal design as possible. The column oriented approach which is used in SSMB (vertical Partitioning, Index only Plans & materialization) will be explained in Section II. This section will also explain decomposition storage Model for implementing Column-Oriented database. Section III will explain the compression techniques to be integrated with Column Oriented database. Section IV will discuss the proposed work and Sec V will explain the conclusion.

2. BACKGROUND AND PRIOR WORK
2.1 Review Stage
The section has three approaches for implementing Column-Store that has been introduced.

2.1.1 Vertical Partitioning
With this approach to make a Column-store in a row-store by partitioning each table vertically [10].An integer value position is added to each column, to connect the fields from

same row together. It consists of more tables with fewer columns. In this way only necessary column are used to respond a query.

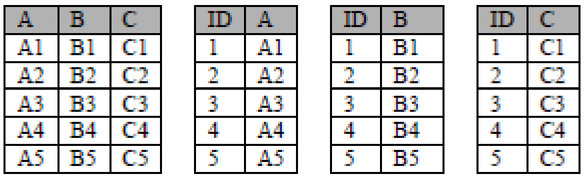
2.1.2 Index-only Plans

Since more tables is to be created with extra position attribute, this leads to wasting more space in vertical partitioning. The alternate approach is index only plans. With this approach an index is to be added for each column of every table and collection of all indices are built so that it is possible to respond a query without ever going to underlying row-oriented tables. The index only plan works by setting list of pairs (surrogate, value) which satisfy the predicate (where clause) in each table.

2.1.3 Materialized View

Using this approach, there is a view with exactly columns needed to respond the query. The main purpose is to create optimal set materialized views where each views is having the required columns to answer the queries**.**

An alternate approach is decomposition storage model. This model vertically partitioned tables [8]. In this model each attribute of table is stored as separate relation along with surrogate (integer value) that identifies the original tuple that the attribute came from. The figure shows sample relation in the NSM representation on far left and the corresponding DSM representation on right [11].



A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3
A4	B4	C4
A5	B5	C5

ID	A
1	A1
2	A2
3	A3
4	A4
5	A5

ID	B
1	B1
2	B2
3	B3
4	B4
5	B5

ID	C
1	C1
2	C2
3	C3
4	C4
5	C5

Fig 2: Example of Decomposition Method

2.2 Decomposition Storage Structure

The DSM model keeps two replica of each partition, one is clustered on IDs as shown above, and the second clustered on attribute value, which is index. DSM exhibit good I/O behavior when the number of attribute is used by query is low projectivity and low selectivity is low. Consider a sample scenario in which a selection operation has low projectivity and low selectivity, i.e. only a few attributes are projected from a large percentage of the tuples. With the DSM representation only the partitions required by the query would be scanned, minimizing the number of disk I/Os performed while maximizing L1 and L2 data cache performance. With the NSM representation, since the query predicate is not very selective, an index would not be useful and the entire table would be scanned [1].

2.3 Fractured Mirror Technique

The other approach is mirroring technique that retains the advantages of both NSM and DSM technique. The queries touching less attribute of large number of records will use .
DSM copy. Queries touching greater part of attributes will use NSM copy. This idea builds on thought of Disk Shadowing [5] [4]. The fractured mirroring technique leads to extra expense on hardware or software.

3. INTEGRATING COMPRESSION WITH COLUMN-SOTRE

Storing uniform type of data in columns presents a number of opportunities for storage size optimizations and also improved performance from compression algorithms. Compression techniques can encode multiple uniform values at once [12]. In row-store such scheme do not work well due to entire tuple belonging to different attribute and data type. Compression algorithms perform better on data with low information entropy (high data value locality) [12][13.]Imagine a database table containing information about customers (name, phone number, e-mail address, e-mail address, etc.). Storing data in columns allows all of the names to be stored together, all of the phone numbers together, etc. Certainly phone numbers will be more similar to each other than surrounding text fields like e-mail addresses or names. Further, if the data is sorted by one of the columns, that column will be super-compressible. Compression is useful because it helps reduce the consumption of expensive resources, such as hard disk space [13].

Row-Store often use dictionary schemes where a dictionary is used to code big values in column into smaller codes e.g. a string-typed column of colors might map “blue” to 0, “yellow” to 1 and “green” to 2 and so on [14] [15] [16] [17]. Sometimes these schemes use prefix coding based on symbol frequencies (e.g. Huffman Coding). In addition to these conventional schemes, Column Store is well suited to compression schemes that compress values from more than one row at a time. This allows large variety of feasible compressions algorithms. E.g. the RLE, where repeats are often expressed as pairs (value, run-length) is attractive approach for compressing sorted data in Column-Oriented database. In this research paper part of section IV we have discussed the how RLE improves the searching in simulation of Column-Store.

4. PROPOSED SOLUTION

In this section we have used the bitmap compression Run Length Encoding (RLE) with simulation of Column-Store, specifically for String operations. We have analyzed the time taken by any query to find out the given string in Compressed and uncompressed form. The simulation psuedocode for compression technique with the searching has entirely written in ‘C++’ language using the Structure

4.1 Run Length Encoding Flow Chart and Psuedocode for proposed Work

The following Figure -3 shows the flow diagram of Run Length Encoding.

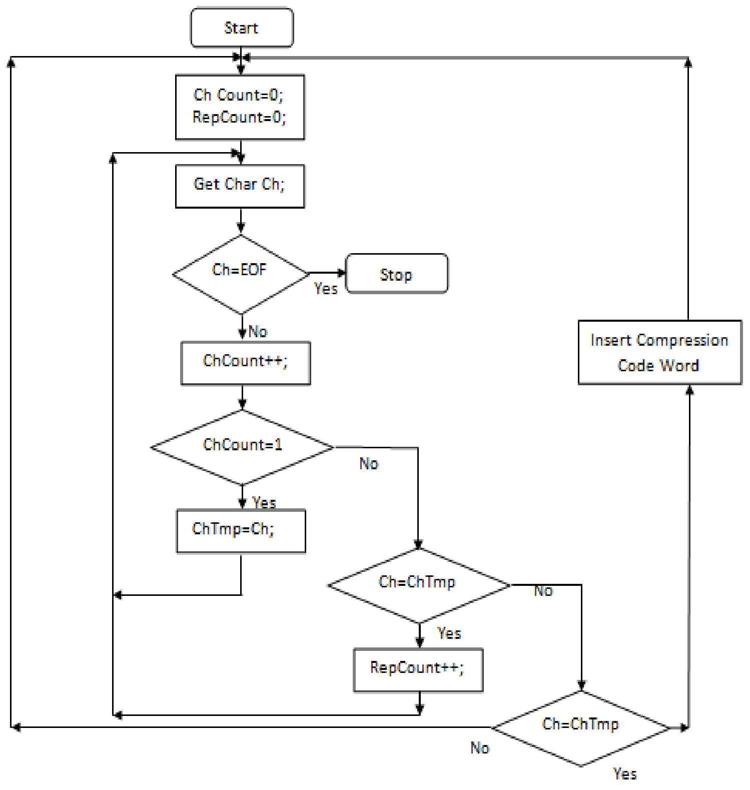


Fig 3: Flow Diagram of RLE Compression

The following is the psuedocode for compression that will be used with or without Compression in searching

Step 1 create required structure

```
Struct cpr
{
char str[100]
char cprstr[50]
}cprarr[250]
```

Step 2 create a subroutine for compression of the string as given below

```
void RLE_Encode(char *str)
{int ii->0
        int ni->0
        int count->1
        char c
        while(*(str+ii)!->'\0')
        {*(str+ni)->*(str+ii)
              ni++
        while(*(str+ii)->->*(str+ii+1))
        {     count++
              ii++
        }
        c->(char)(((int)'0')+count)
              *(str+ni)->c
               ni++
               ii++
              count->1
              }*(str+ni)->'\0'}
```

Step 4 Create a subroutine for searching with custom filters technique as given below

```
float search(int op1)
bool found=false;
float ans->0.0f
clock_t t1,t2  //used to store system current time
if(op1==2)
Input string to be search in 'item'
RLE_Enocode(item) start compressing the input before comparison
endif
else
Input string to be search in 'item'
t1->clock()
for(i->0 i<250 i++)
{ compare and check if cpratt[i].cpr with item || op==2 and cprarr[i].cprpd with item
then
found=true}
t2->clock() }
return (t2-t1)
   }
```

4.1.1 Algorithm for Proposed Work

1. First create a columnar database
2. Read the required column needed to compress the data
3. Store the compressed data in another column
4. Apply Searching on compressed and uncompressed data
5. Determine the time difference with/without compression.

4.1.2 Experiment & Result

Execution of the above code and the experiment were carried out on windows platform with 2.2 GHz processor and 4GB RAM. We have built one dimensional table through the structure in C++ programming language and stored 250 records and applied Run Length Encoding (RLE). Henceforth we have concluded with the following decision table -1 and fig-3.

Table 1. Comparative tabular data for RLE Compression

Number Of Records	Time Without RLE (T1)	Time with RLE (T2)	Time Diff. (T1-T2)	%age of Efficiency
10	9.21	6.1	3.11	33.77
20	17.03	10.38	6.65	39.05
50	29.62	19.54	10.08	34.03
100	18.97	8.34	10.63	56.04
250	79.59	42.97	36.62	46.01

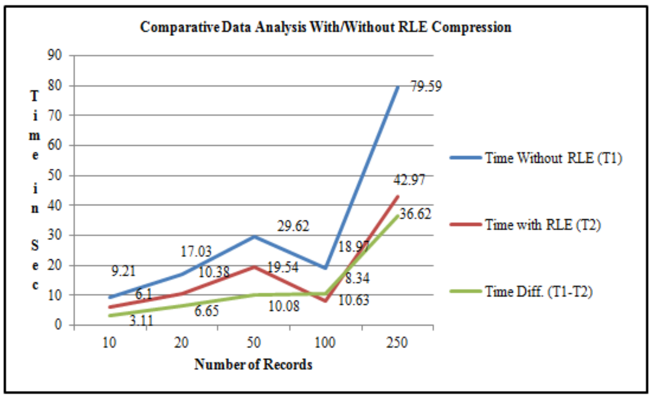


Fig 4: Comparative Data Analysis With/Without RLE

5. CONCLUSION

The inclination of proposed work shows that the significant database performance gains can be kept by implementing the Optimal Compression schemes that work directly on compressed data. Furthermore our focus on column oriented compression allowed us to optimize the storage space and enhance the searching efficiency in the column-store is greater than row-store. Through this work the time efficiency is increase by 43.42% and storage space has been reduced by almost 50%. Hence we observe through this research work as a significant role in understanding the considerable performance gains of Column-oriented design while this paper centered on quite simple query so as to refine the performance edge of column-oriented compression

6. ACKNOWLEDGMENTS

First of all, I express my sincere gratitude to the Almighty Allah. I am very great full to my advisor Dr Shish Ahmad and for his encouragement, and many fruitful discussions from the very early stage of this project. I would like to thank Dr (Prof) Rizwan Beg, Former Head of Department of Computer Science and Engineering, Integral University, Lucknow. I would like to thanks Mrs. Kavita Agarwal (HOD), for her great help. Finally, I wish to express my appreciation to my family for my continuous love and encouragement, for always believing in me, for never failing to provide all the support, and for coping with the pressure that naturally comes with such endeavor.

5. REFERENCES

[1] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, M. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Artitectures. 2012.W.-K. Chen, Linear Networks and Systems. Belmont, Calif.: Wadsworth, pp. 123-135, 1993. (Book style)

[2] P. Boncz, M. Zukowski, N. Nes. MonetDB/X100: Hyper-pipeliningquery execution. In CIDR, 2005K. Elissa, “An Overview of Decision Theory," unpublished. (Unplublished manuscript)

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

[4] http:///www.Data Business Intelligence and Column Database Technology/InfinitDB/by Calpont.mht

[5] Column-oriented DBMS-Wikipedia, the free encyclopedia.mht

[6] Gheorghe MATEI: Column-Oriented Databases, an Alternative for Analytical Environment

[7] Data/HRG/Home.htm

[8] D.J. Abadi, S.R. Madden, N. Hachem. Column-stores vs. row-stores: how different are they really? In Proc. SIGMOD, 2008.

[9] P. E. O’Neil, X. Chen, E. J. O’Neil. Adjoined Dimension Column Index (ADC Index) to Improve Star Schema Query Performance. In ICDE, 2008

[10] P. E. O’Neil, E. J. O’Neil, X. Chen. The Star Schema Benchmark (SSB). http: //www.cs.umb.edu

[11] G.P.Copeland, S.Khosafian. A Decomposition Storage Model. Proceedingsof ACM SIGMOD 1985.

[12] Daniel J.Abadi, Samuel R Madden,Miguel C.Ferreira.:Integrating Compression and Execution in Column –Oriented Database Systems

[13] Daniel J.Abadi, Samuel R Madden,Miguel C.Ferreira.:Integrating Compression and Execution in Column –Oriented Database Systems

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

[15] M. A. Roth and S. J. V. Horn. Database compression. SIGMOD Rec., 22(3):31{39, 1993.

[16] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. In SIGMOD '01, Pages 271- 282, 2001

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

[18] Shish Ahmad. Evaluation of security risk associated with different network layers' published in International Journal of computer application Jul 2012