An Analysis on efficient column based storage structures in Big Data Analytics

J V N Lakshmi*
Research Scholar
Department of Research in Computer Science
SCSVMV University, Kanchipuram – INDIA
jlakshmi.research@gmail.com

Dr. Ananthi Sheshasayee
Research Supervisor
Department of Computer Science & Research
Quaid – E – Millath Govt College for Women
Chennai – INDIA
ananthi.research@gmail.com

Abstract

Map Reduce-based data warehouse systems are playing important roles of supporting big data analytics to understand quickly the dynamics of user behavior trends and their needs in typical Web service providers and social network sites (e.g., Facebook). In such a system, the data placement structure is a critical factor that can affect the warehouse performance in a fundamental way. Based on analysis a system requires the data placement structure characterized as: (1) fast data loading, (2) highly efficient storage space utilization, and (3) strong adaptivity to highly dynamic workload patterns. This paper examines different table structures implemented on HDFS using Map Reduce model such as Table Placement Method, SLCG store, RC File, Hbase, Trojan Layout and Column store. Observation analyses that column based stores proves to be more sophisticated compared to row store**.**

Key Words **—** Column Store, Data Analytics, HBase, Map Reduce, Hadoop, RC File, Row Store, Trojan Layout

1. **Introduction**

This is a data explosion era, where data is increasing exponentially. Such massive amount of data is called Big Data. The data so produced needs certain analysis, processing data management and huge storage capacity [10]. Analyzing terabytes of data is a common task for many enterprises such as Social networking sites, Government organizations, Enterprises and Technical agencies. To manage this trend, Map Reduce is quickly becoming the de facto standard for large-scale analysis in industry [1].

These Map Reduce-based warehouse systems cannot directly control storage disks in clusters. Instead, they have to utilize the cluster-level distributed file system (e.g. HDFS, the Hadoop Distributed File System) to store a huge amount of table data [9]. Therefore, a serious challenge in building such a system is to find an efficient data placement structure that determines how to organize table data in the underlying HDFS [14].

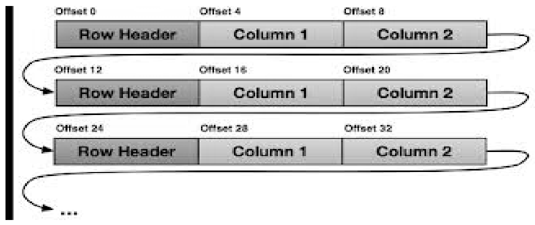

Fig 1: AN EXAMPLE OF ROW-STORE Src: [7]

The traditional data warehousing system uses a row – store structure which dominates in conventional one-size fits- all database systems [16]. With this structure, relational records are organized in an N-array storage model [8]. All fields of one record are padded one by one in the order of their occurrences. Records are placed contiguously in a disk page.

In this paper section 2 discuss on related literature. Section 3 talks about various Column based table structures. The section 4 examines comparison with row based structure with column oriented structures. Section 5 concludes.

2. Related Literature

The data load time overhead of Trojan HDFS is negligible. Furthermore, the one-time data load cost of Trojan HDFS pays of as recurring speed-ups over several Map Reduce jobs. Trojan Layouts can create a different layout per data block replica still table structure is not efficient when compared to column stores.

A column store heavily employs compression, though only Column Store offers compressed execution. C-Store follows the idea of storing data multiple times in projections with a different sort order. In table structure efficiency and data stores have a high feasibility.

RCFile uses two sections to store the real data of each column and the metadata about this column (mainly the length of each cell), and compress the two sections independently. Thus, RCFile has better data compression efficiency. Column-store and column-group has significantly long loading times than both row-store and RCFile.

Mastiff's SLC-Store has a much larger horizontal partition granularity than RCFile and in each horizontal partition SLC-Store uses a column group store while RCFile uses a pure column store. Evaluations with various workloads show that Mastiff is up to 5 times faster in data loading and up to 7 times faster in aggregate query execution than other systems including Hive/RCFile.

3. Column based storage structures

3.1 Column Store

Column-stores - In recent years, there has been renewed interest in so-called *column-oriented systems*, sometimes also called *column-stores* [3]. Column-store systems completely vertically partition a database into a collection of individual columns that are stored separately. By storing each column separately on disk, these column-based systems enable queries to read just the attributes they need, rather than having to read entire rows from disk and discard unneeded attributes in memory [6].

A similar benefit is true while transferring data from main memory to CPU registers, improving the overall utilization of the available I/O and memory bandwidth [15]. Overall, taking the column-oriented approach to the extreme allows for numerous innovations in terms of database architectures. Modern column-stores, their architecture and evolution as well the benefits they can bring in data analytics.

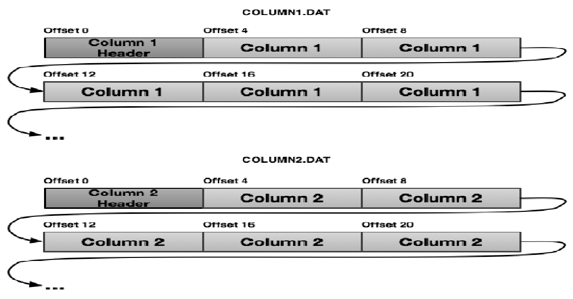

Fig 2: A COLUMN STORE Src: [7]

In the column-oriented approaches each column is stored independently as a separate data object. Since data is typically read from storage and written in storage in blocks, a column-oriented approach means that each block which holds data for the table holds data for one of the columns. The data need to access the provided date columns, and the data blocks corresponding to these columns would need to be read from storage [3].

3.2 Rc File – Record Columnar File

Column-store can avoid reading unnecessary columns during a query execution, and can easily achieve a high compression ratio by compressing each column within the same data domain. Column-store cannot guarantee that all fields in the same record are located in the same cluster node. Therefore, a record reconstruction will cause a large amount of data transfers via networks among multiple cluster nodes.

RCFile is designed and implemented on top of the Hadoop Distributed File System (HDFS). As demonstrated in the above Figure 3 as shown. RCFile has the following data layout to store a table:

1. According to the HDFS structure, a table can have multiple HDFS blocks.
2. In each HDFS block, RCFile organizes records with the basic unit of a *row group*. That is to say, all the records stored in an HDFS block are partitioned into row groups. For a table, all row groups have the same size.

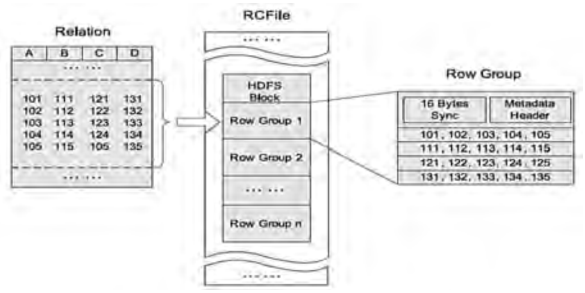

Fig 3: RC FILE TABLE STRUCTURE Src : [4]

3. A row group contains three sections. The first section is a sync marker that is placed in the beginning of the row group. The sync marker is mainly used to separate two continuous row groups in an HDFS block. The second section is a Metadata header for the row group. The third section is the table data section that is actually a column-store [4].

3.3 Trojan Layout

The core idea of per-replica Trojan Layout is to first create query groups and then create column groups for each query group separately [1]. This serves two purposes:

(i). Instead of creating a single layout for the entire workload, create multiple layouts; each specialized for a part of the workload.

(ii). Query grouping can significantly decrease the number of referenced attributes for each query group reduces the complexity of our column grouping store.

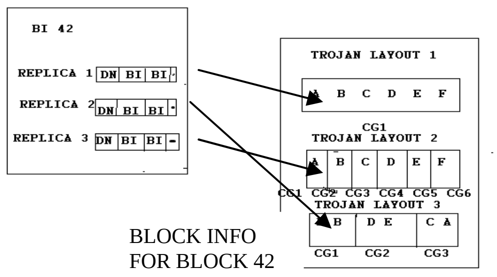

Figure 4: Quadruplets for a data block in Trojan HDFS stored at the name node. This structured is composed: (i) of a pointer to the data node (e.g. DN 7) storing a replica (e.g. the first replica) of a data block (e.g. data block 42), (ii) of a pointer to the previous data block (e.g. data block 21) stored on DN 7, (iii) of a pointer to the next block (e.g. data block 51) stored on DN 7, and (iv) of a pointer to the Trojan Layout descriptor for that data block replica

Fig 4: TROJAN LAYOUT COLUMN GROUPING Src: [11]

Implementation of a variant of HDFS, called Trojan HDFS, to introduce per-replica Trojan Layouts into HDFS. Trojan HDFS differs from HDFS in two aspects:

- The name node in Trojan HDFS keeps a catalog of the Trojan Layouts of all data block replicas. Trojan HDFS exploits the fact that the name node maintains a triplet of pointers for each data block points to the Trojan Layout descriptor of the data block replica. Figure 5 illustrates this quadruplet of pointers associated to a data block replica. Note that more than one data block replica could point to the same Trojan Layout descriptor [1].
- A data node in Trojan HDFS asks the name node for the Trojan Layout of each data block replica stored locally. After receiving the Trojan Layout for a given data block replica, a data node internally reorganizes the data of the data block replica according to the received layout. There are two ways: (i) reorganize a data block as soon as the data block replica is copied locally, or (ii) reorganize a data block after all replicas of the data block are copied to relevant data nodes.

3.4 Hbase – Schemaless Database

HBase is a database that sits on top of the HDFS and has tight Map Reduce integration. Like HDFS and Map Reduce, it’s based on a technology described in a Google paper; that technology is called Big Table [13].

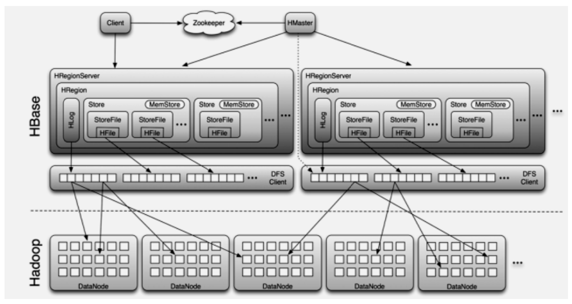

FIG 5: HBASE REGION STRUCTURE Src : [2]

Just like the databases, HBase exposes a data model consisting of tables containing rows, with data within those rows organized into columns. That's basically where the similarities end [2]. From the web page, it's designed to manage *big* tables, billions of rows with millions of columns. HBase is a "schemaless" database says that ahead of time what columns a table will contain.

3.5 Mastiff – Slcg Store

A structured data store over HDFS called Segment-Level Column Group Store (SLC-Store), which is a RCFile-like column store integrated with two kinds of light-weight helper structures, to improve both data loading speed and analytical query performance.

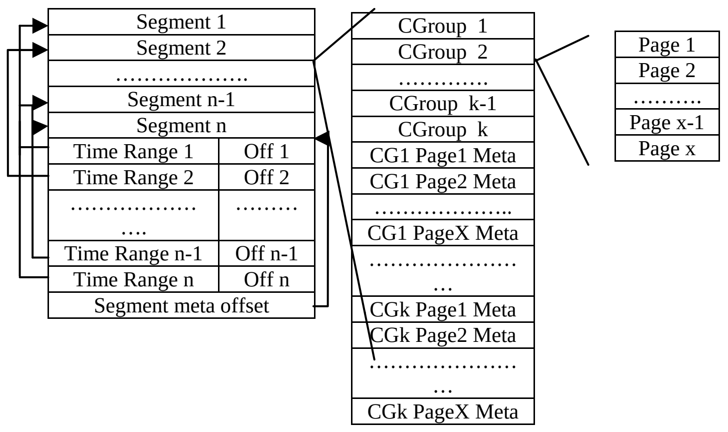

FIG 6: SEGMENT LEVEL COLUMN GROUP STORE IN MASTIFF Src: [5]

As shown in Figure 6, a table in Mastiff is first horizontally partitioned into multiple segments. In each segment, data are vertically partitioned into column groups, and each group may consist of one or more columns. Mastiff table is physically stored by a single HDFS file and each segment is stored in an HDFS block. All column groups in the same segment are stored one by one. Each segment is physically divided into fixed-size pages and each column group in the segment is stored in contiguous pages [5].

Mastiff keeps two light-weight helper structures on disk along with the data. The segment-level helper structure keeps the time range and the offset of each segment, and is stored at the end of each file. The segment-level helper structure is a kind of coarse-grained helper structure, which reduces only the number of Map tasks. The page-level helper structure in Mastiff is a kind of finer-grained helper structure. It is co-located with data, unlike many indexes which are stored in separate files [5].

3.6 Table Placement Method

This section defines the basic structure of table placement methods. The basic structure of a table placement method comprises three consecutive procedures, a row reordering procedure, a table partitioning procedure, and a data packing procedure [8].

These three procedures are represented by three corresponding functions, which are f_{RR}, f_{TP}, and f_{DP}, respectively. In our definition, all rows of a table form a row sequence. The position of a specific row in the row sequence refers to this row, e.g. the first row. Also, all columns of a table form a column sequence. The position of a specific column in the column sequence refers to this column, e.g. the second column. In this way, the use of position refers to a specific data value in the table, e.g. the data value at the first row and the second column [8].

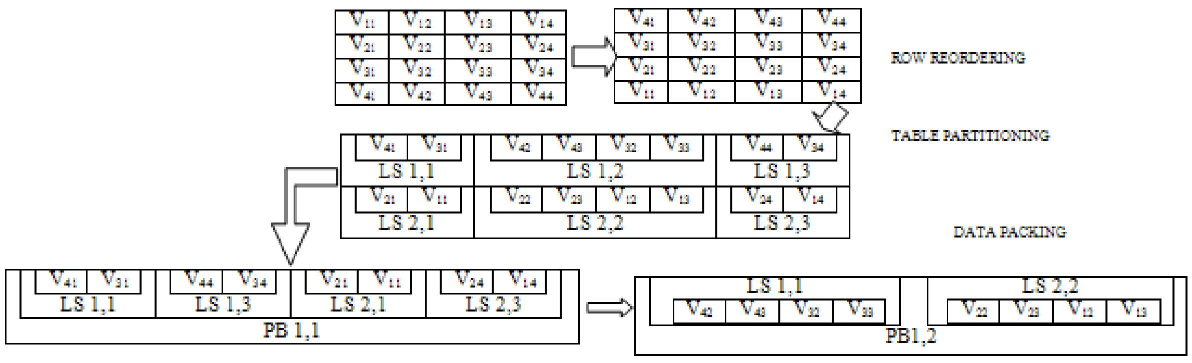

Fig 7: TABLE PLACEMENT METHOD Src: [8]

4. Comparisons among various table structures

The comparison of various table structures basing on parameters data loading, data compression, data storage and table structures.

A. *Data Loading:* For a data placement structure, data loading time is an important factor for daily operations. Reducing this time is critical for processing 20TB data into the production warehouse every day.

B. *Data Compression:* Compression algorithms perform better on data with low information entropy (i.e., with high data value locality), and values from the same column tend to have more value locality than values from different columns.

C. *Data Storage:* The storage space sizes required by the raw data and several data placement structures vary. Data compression can significantly reduce the storage space, and different data placement structures show different compression efficiencies.

TABLE 1: COMPARISONS OF VARIOUS TABLE STRUCTURES

Parameters	**Table Structures**						
	Row store	Column Store	RC File	Trojan Layout	HBase	Mastiff	Table Placement Method
Data Loading	35%	45%	61%	43%	64%	67%	56%
Data Compression / Data Packing	23%	56%	63%	54%	57%	65%	45%
Data Storage	39%	61%	69%	65%	72%	71%	66%
Table Structure Efficiency	26%	63%	67%	42%	69%	70%	54%
Type of Table Structure	Row	Column	Records	Column Group	Row-Column	Segment Column Group	Grouping Columns

D. T*able Structure:* Row-store efficiency is compared with various table structures Trojan layout, HBase, Table Placement Method, and SLCG store, column store, and RCFile.

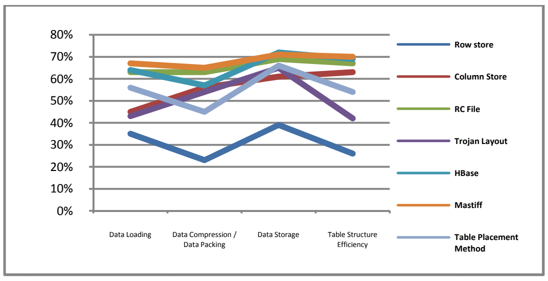

FIG 8: PARAMETERS COMPARISON ON TABLE STRUCTURES

The table1 compares the various table structures with row stores by viewing on different parameters. The results on comparison reveals that the column oriented data structures have fast data loading, high storage and data compression than row stores from fig 8.

5. Conclusion

The Row Store has less compression efficiency when compared to column stores. The table structure organized in RCFile has high data compression. Data loading in SLCG store of Mastiff has better performance. The inference from comparison Table Placement method's row group size should be large enough so that the column (or column group) size inside a row group will be large enough and when the row group size is large enough and the column-oriented access method is used to read columns, it is not necessary to group multiple columns to a column group. So, therefore the combinations of row with multiple column groups will definitely have an effective impact on data storage structures.

References

[1] "Trojan Data Layouts: Right Shoes for a Running Elephant", by Alekh Jindal, Jorge-Arnulfo Quiané-Ruiz, Jens Dittrich
[2] "Hbase Schema Design nosq"l Cologne, April 2013 Lars George Director EMEA Services.
[3] "The Design and Implementation of Modern Column-Oriented Database Systems" , by Daniel Abadi , Peter Boncz , Stavros Harizopoulos
[4] "Rcfile: A Fast and Space-efficient Data Placement Structure in mapreduce-based Warehouse Systems", by Yongqiang He, Rubao Lee, Yin Huai
[5] "Mastiff: A mapreduce-based System for Time-based Big Data Analytics" , by Sijie Guo, Jin Xiong, Weiping Wang Rubao Lee
[6] J. Abadi, D. S. Myers, D. J. Dewitt, and S. Madden, "Materialization strategies in a column-oriented dbms," ICDE, 2007.
[7] M. Stonebraker, D. J. Abadi, P. E. O'Neil, A. Rasin, N. Tran, and "C-store: A column-oriented dbms," in VLDB, 2005.
[8] Chang, J. Dean, S. Ghemawat, A. Fikes, "Bigtable: A distributed storage system for structured data," in OSDI, 2006, pp. 205–218.
[9] Http://hbase.apache.org. and Http://mapreduce.apache.org
[10] J. Abadi, s. Madden, and n. Hachem, "Column-stores vs. Row-stores: how different are they really?" In sigmod conference, 2008
[11] Bentley, j. L., and mcilroy, m,"Data compression using long common strings". In data compression conference (1999), pp. 287.295.
[12] Bloom, b. H. Space/time trade-offs in hash coding with allowable errors. Cacm 13, 7 (1970), 422.426.
[13] Abouzeid, Abadi, A. Rasin, "Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads,"PVLDB, 2009
[14] S. Navathe et al. "Vertical Partitioning Algorithms for Database Design". ACM TODS, 9(4):680–710, 1984.