RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems

Yongqiang He [#$1], Rubao Lee [%2], Yin Huai [%3], Zheng Shao [#4], Namit Jain [#5], Xiaodong Zhang [%6], Zhiwei Xu [$7]

[#]*Facebook Data Infrastructure Team*
{[1]heyongqiang, [4]zshao, [5]njain}@fb.com

[%]*Department of Computer Science and Engineering, The Ohio State University*
{[2]liru, [3]huai, [6]zhang}@cse.ohio-state.edu

[$]*Institute of Computing Technology, Chinese Academy of Sciences*
[7]zxu@ict.ac.cn

Abstract— **MapReduce-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 our observations and analysis of Facebook production systems, we have characterized four requirements for the data placement structure: (1) fast data loading, (2) fast query processing, (3) highly efficient storage space utilization, and (4) strong adaptivity to highly dynamic workload patterns. We have examined three commonly accepted data placement structures in conventional databases, namely row-stores, column-stores, and hybrid-stores in the context of large data analysis using MapReduce. We show that they are not very suitable for big data processing in distributed systems. In this paper, we present a big data placement structure called RCFile (Record Columnar File) and its implementation in the Hadoop system. With intensive experiments, we show the effectiveness of RCFile in satisfying the four requirements. RCFile has been chosen in Facebook data warehouse system as the default option. It has also been adopted by Hive and Pig, the two most widely used data analysis systems developed in Facebook and Yahoo!**

I. Introduction

We have entered an era of data explosion, where many data sets being processed and analyzed are called "big data". Big data not only requires a huge amount of storage, but also demands new data management on large distributed systems because conventional database systems have difficulty to manage big data. One important and emerging application of big data happens in social networks on the Internet, where billions of people all over the world connect and the number of users along with their various activities is growing rapidly. For example, the number of registered users in Facebook, the largest social network in the world has been over 500 million [1]. One critical task in Facebook is to understand quickly the dynamics of user behavior trends and user needs based on big data sets recording busy user activities.

The MapReduce framework [2] and its open-source implementation Hadoop [3] provide a scalable and fault-tolerant infrastructure for big data analysis on large clusters. Furthermore, MapReduce-based data warehouse systems have been successfully built in major Web service providers and social network Websites, and are playing critical roles for executing various daily operations including Web click-stream analysis, advertisement analysis, data mining applications, and many others. Two widely used Hadoop-based warehouse systems are Hive [4][5] in Facebook and Pig [6] in Yahoo!

These MapReduce-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. 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. Being a critical factor that can affect warehouse performance in a fundamental way, such a data placement structure must be well optimized to meet the big data processing requirements and to efficiently leverage merits in a MapReduce environment.

A. Big Data Processing Requirements

Based on our analysis on Facebook systems and huge user data sets, we have summarized the following four critical requirements for a data placement structure in a MapReduce environment.

1) *Fast data loading*. Loading data quickly is critical for the Facebook production data warehouse. On average, more than 20TB data are pushed into a Facebook data warehouse every day. Thus, it is highly desirable to reduce data loading time, since network and disk traffic during data loading will interfere with normal query executions.
2) *Fast query processing*. Many queries are response-time critical in order to satisfy the requirements of both real-time Website requests and heavy workloads of decision supporting queries submitted by highly-concurrent users. This requires that the underlying data placement structure retain the high speed for query processing as the amount of queries rapidly increases.
3) *Highly efficient storage space utilization*. Rapidly growing user activities have constantly demanded scalable storage capacity and computing power. Limited disk

space demands that data storage be well-managed, in practice, to address the issues on how to place data in disks so that space utilization is maximized.

4) *Strong adaptivity to highly dynamic workload patterns*. Data sets are analyzed by different application users for different purposes in many different ways [7]. Some data analytics are routine processes that are executed periodically in a static mode, while some are ad-hoc queries issued from internal platforms. Most workloads do not follow any regular patterns, which demand the underlying system be highly adaptive to unexpected dynamics in data processing with limited storage space, instead of being specific to certain workload patterns.

B. Data Placement for MapReduce

A critical challenge in designing and implementing an efficient data placement structure for a MapReduce-based data warehouse system is to address the above four requirements considering unique features in a MapReduce computing environment. In conventional database systems, three data placement structures have been widely studied, which are:

1) *horizontal row-store structure* ([8]),
2) *vertical column-store structure* ([9][10][11][12]), and
3) *hybrid PAX store structure* ([13][14]).

Each of these structures has its advantages considering one of the above requirements. However, simply porting these database-oriented structures into a MapReduce-based data warehouse system cannot fully satisfy all four requirements.

We here briefly summarize major limitations of these structures for big data, and will provide a detailed evaluation on the three structures in Section II. First, row-store cannot support fast query processing because it cannot skip unnecessary column reads when a query only requires only a few columns from a wide table with many columns [10]. Second, column-store can often cause high record reconstruction overhead with expensive network transfers in a cluster. This is because, with column-store, HDFS cannot guarantee that all fields in the same record are stored in the same cluster node. Although pre-grouping multiple columns together can reduce the overhead, it does not have a strong adaptivity to respond highly dynamic workload patterns. Third, with the goal of optimizing CPU cache performance, the hybrid PAX structure that uses column-store inside each disk page cannot help improve the I/O performance [15][12] for analytics of big data.

In this paper, we present our data placement structure, called RCFile *(Record Columnar File)*, and its implementation in Hadoop. We highlight the RCFile structure as follows.

1) A table stored in RCFile is first horizontally partitioned into multiple *row groups*. Then, each row group is vertically partitioned so that each column is stored independently.
2) RCFile utilizes a column-wise data compression within each row group, and provides a *lazy decompression* technique to avoid unnecessary column decompression during query execution.
3) RCFile allows a flexible row group size. A default size is given considering both data compression performance and query execution performance. RCFile also allows users to select the row group size for a given table.

We have implemented RCFile in Hadoop. As to be discussed and evaluated in this paper, RCFile can satisfy all the above four requirements, since it not only retains the merits of both the row-store and the column-store, and also has added PAX's missing role of I/O performance improvement. After extensive evaluation in production systems, RCFile has been chosen in the Facebook Hive data warehouse system as the default option. Besides, RCFile has also been adopted in the Yahoo Pig system, and is being considered in other MapReduce-based data processing systems.

The rest of the paper is organized as follows. We present a detailed analysis of existing three data placement structures in Section II. We present the design, implementation, and several critical issues of our solution RCFile in Section III. In Section IV, we introduce the API of RCFile. Performance evaluation is presented in Section V. We overview other related work in Section VI, and conclude the paper in Section VII.

II. Merits and Limitations of Existing Data Placement Structures

In this section, we will introduce three existing data placement structures, and discuss why they may not be suitable to a Hadoop-based data warehouse system.

A. Horizontal Row-store

The row-store structure dominates in conventional one-size-fits-all database systems [16]. With this structure, relational records are organized in an N-ary 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. Figure 1 gives an example to show how a table is placed by the row-store structure in an HDFS block.

The major weaknesses of row-store for read-only data warehouse systems have been intensively discussed. First, row-store cannot provide fast query processing due to unnecessary column reads if only a subset of columns in a table are needed in a query. Second, it is not easy for row-store to achieve a high data compression ratio (and thus a high storage space utilization) due to mixed columns with different data domains

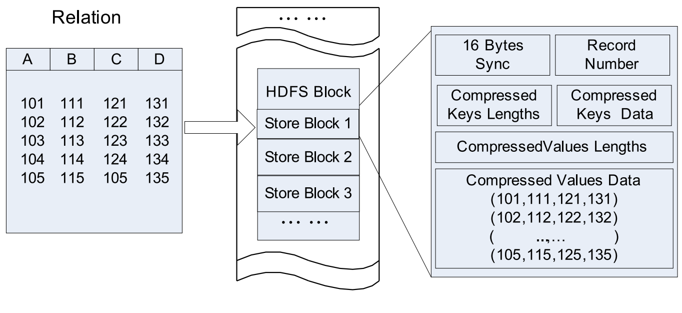


Fig. 1: An example of row-store in an HDFS block.

[10][11]. Although prior work [17][18] has shown that row-store with careful *entropy coding* and the utilization of *column correlations* can have a better data compression ratio than that of column-store, it can cause a high data decompression overhead by complex data storage implementations.

The major advantage of row-store for a Hadoop-based system is that it has fast data loading and strong adaptive ability to dynamic workloads. This is because row-store guarantees that all fields in the same record is located in the same cluster node since they are in the same HDFS block.

B. Vertical Column-store

The vertical store scheme is based on a column-oriented store model for read-optimized data warehouse systems. In a vertical storage, a relation is vertically partitioned into several sub-relations. Basically, there are two schemes of vertical stores. One scheme is to put each column in one sub-relation, such as the Decomposition Storage Model (DSM) [9], MonetDB [19], and an experimental system in [15]. Another scheme is to organize all the columns of a relation into different column groups, and usually allow column overlapping among multiple column groups, such as C-store [10] and Yahoo Zebra [20]. In this paper, we call the first scheme *the column-store*, and the second one *the column-group*. Notice that, for column-group, how data is organized (row-oriented or column-oriented) in a group is dependent on system implementations. In C-store, a column group uses the column-store model, i.e., each column is stored individually. However, to accelerate a record reconstruction activity, all columns in a column group must be ordered in the same way. In Zebra used for the Pig System, a column group is actually row-oriented to reduce the overhead of a record reconstruction in a MapReduce environment.

Figure 2 shows an example on how a table is stored by column-group on HDFS. In this example, column A and column B are stored in the same column group, while column C and column D are stored in two independent column groups.

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. However, it cannot provide fast query processing in Hadoop-based systems due to high overhead of a tuple reconstruction. Column-store cannot guarantee that all fields in the same record are located in the same cluster node. For instance, in the example in Figure 2, the four fields of a record are stored in three HDFS blocks that can be located in different nodes. Therefore, a record reconstruction will cause a large amount of data transfers via networks among multiple cluster nodes. As introduced in the original MapReduce paper [2], excessive network transfers in a MapReduce cluster can always be a bottleneck source, which should be avoided if possible.

Since a column group is equivalent to a materialized view, it can avoid the overhead of a record reconstruction [10]. However, it cannot satisfy the requirement of quickly adapting dynamic workloads, unless all column groups have been created with the pre-knowledge of possible queries. Otherwise, for a query that needs a non-existing combination of columns, a record reconstruction is still required to use two or more existing column groups. Furthermore, column-group can also create redundant column data storage due to the column overlap among multiple groups. This would under-utilize storage space.

C. Hybrid Store: PAX

PAX [13] and its improvement in Data Morphing [21] use a hybrid placement structure aiming at improving CPU cache performance. For a record with multiple fields from different columns, instead of putting these fields into different disk pages, PAX puts them in a single disk page to save additional operations for record reconstructions. Within each disk page, PAX uses a mini-page to store all fields belonging to each column, and uses a page header to store pointers to mini-pages.

Like row-store, PAX has a strong adaptive ability to various dynamic query workloads. However, since it was mainly proposed for performance improvement of CPU cache utilization for data sets loaded in main memory, PAX cannot directly satisfy the requirements of both high storage space utilization and fast query processing speed on large distributed systems for the following three reasons.

1) PAX is not associated with data compression, which is not necessary for cache optimization, but very important for large data processing systems. It does provide an opportunity to do column-wise data compression [13].
2) PAX cannot improve I/O performance because it does not change the actual content of a page [15][12]. This

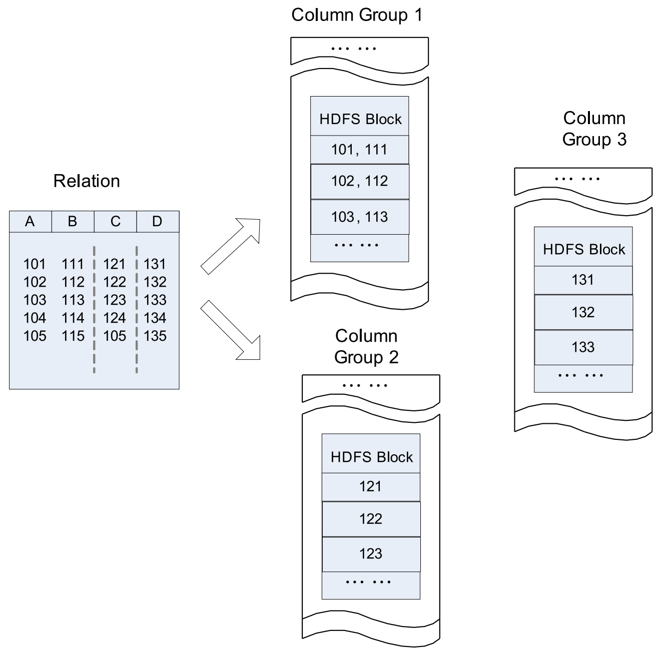


Fig. 2: An example of column-group in an HDFS block. The four columns are stored into three column groups, since column A and B are grouped in the first column group.

limitation would not help our goal of fast query processing for a huge amount of disk scans on massively growing data sets.

3) Limited by the page-level data manipulation inside a traditional DBMS engine, PAX uses a fixed page as the basic unit of data record organization. With such a fixed size, PAX would not efficiently store data sets with a highly-diverse range of data resource types of different sizes in large data processing systems, such as the one in Facebook.

III. The Design and Implementation of RCFile

In this section, we present RCFile (Record Columnar File), a data placement structure designed for MapReduce-based data warehouse systems, such as Hive. RCFile applies the concept of *"first horizontally-partition, then vertically-partition"* from PAX. It combines the advantages of both row-store and column-store. First, as row-store, RCFile guarantees that data in the same row are located in the same node, thus it has low cost of tuple reconstruction. Second, as column-store, RCFile can exploit a column-wise data compression and skip unnecessary column reads.

A. Data Layout and Compression

RCFile is designed and implemented on top of the Hadoop Distributed File System (HDFS). As demonstrated in the example shown in Figure 3, 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. Depending on the row group size and the HDFS block size, an HDFS block can have only one or multiple row groups.

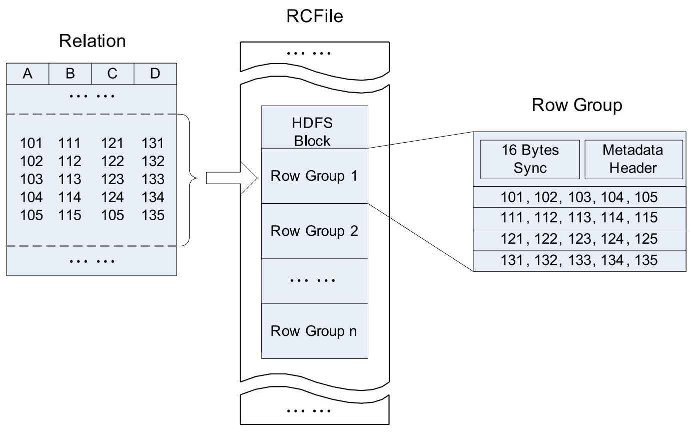


Fig. 3: An example to demonstrate the data layout of RCFile in an HDFS block.

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 metadata header stores the information items on how many records are in this row group, how many bytes are in each column, and how many bytes are in each field in a column. The third section is the table data section that is actually a column-store. In this section, all the fields in the same column are stored continuously together. For example, as shown in Figure 3, the section first stores all fields in column A, and then all fields in column B, and so on.

We now introduce how data is compressed in RCFile. In each row group, the metadata header section and the table data section are compressed independently as follows.

- First, for the whole metadata header section, RCFile uses the RLE (Run Length Encoding) algorithm to compress data. Since all the values of the field lengths in the same column are continuously stored in this section, the RLE algorithm can find long runs of repeated data values, especially for fixed field lengths.
- Second, the table data section is not compressed as a whole unit. Rather, each column is independently compressed with the Gzip compression algorithm. RCFile uses the heavy-weight Gzip algorithm in order to get better compression ratios than other light-weight algorithms. For example, the RLE algorithm is not used since the column data is not already sorted. In addition, due to the lazy decompression technology to be discussed next, RCFile does not need to decompress all the columns when processing a row group. Thus, the relatively high decompression overhead of the Gzip algorithm can be reduced.

Though currently RCFile uses the same algorithm for all columns in the table data section, it allows us to use different algorithms to compress different columns. One future work related to the RCFile project is to automatically select the best compression algorithm for each column according to its data type and data distribution.

B. Data Appending

RCFile does not allow arbitrary data writing operations. Only an appending interface is provided for data writing in RCFile because the underlying HDFS currently only supports data writes to the end of a file. The method of data appending in RCFile is summarized as follows.

1) RCFile creates and maintains an in-memory *column holder* for each column. When a record is appended, all its fields will be scattered, and each field will be appended into its corresponding column holder. In addition, RCFile will record corresponding metadata of each field in the metadata header.
2) RCFile provides two parameters to control how many records can be buffered in memory before they are

flushed into the disk. One parameter is the limit of the number of records, and the other parameter is the limit of the size of the memory buffer.

3) RCFile first compresses the metadata header and stores it in the disk. Then it compresses each column holder separately, and flushes compressed column holders into one row group in the underlying file system.

C. Data Reads and Lazy Decompression

Under the MapReduce framework, a mapper is started for an HDFS block. The mapper will sequentially process each row group in the HDFS block.

When processing a row group, RCFile does not need to fully read the whole content of the row group into memory. Rather, it only reads the metadata header and the needed columns in the row group for a given query. Thus, it can skip unnecessary columns and gain the I/O advantages of column-store. For instance, suppose we have a table with four columns $tbl(c_1, c_2, c_3, c_4)$, and we have a query *"SELECT* c_1 *FROM tbl WHERE* $c_4 = 1$*"*. Then, in each row group, RCFile only reads the content of column c_1 and c_4.

After the metadata header and data of needed columns have been loaded into memory, they are all in the compressed format and thus need to be decompressed. The metadata header is always decompressed and held in memory until RCFile processes the next row group. However, RCFile does not decompress all the loaded columns. Instead, it uses a lazy decompression technique.

Lazy decompression means that a column will not be decompressed in memory until RCFile has determined that the data in the column will be really useful for query execution. Lazy decompression is extremely useful due to the existence of various where conditions in a query. If a where condition cannot be satisfied by all the records in a row group, then RCFile does not decompress the columns that do not occur in the where condition. For example, in the above query, column c_4 in any row group must be decompressed. However, for a row group, if no field of column c_4 in the group has the value 1, then it is unnecessary to decompress column c_1 in the group.

D. Row Group Size

I/O performance is a major concern of RCFile. Therefore, RCFile needs to use a large and flexible row group size. There are two considerations to determine the row group size:

1) A large row group size can have better data compression efficiency than that of a small one. However, according to our observations of daily applications in Facebook, when the row group size reaches a threshold, increasing the row group size cannot further improve compression ratio with the Gzip algorithm.
2) A large row group size may have lower read performance than that of a small size because a large size can decrease the performance benefits of lazy decompression. Furthermore, a large row group size would have a higher memory usage than a small size, and would affect executions of other co-running MapReduce jobs.

1) Compression efficiency: In order to demonstrate how different row group sizes can affect the compression ratio, we have conducted an experiment with a portion of data from a table in Facebook. We examined the table sizes using different row group sizes (from 16KB to 32MB), and the results are shown in Figure 4.

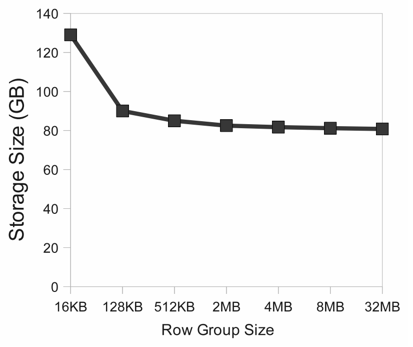


Fig. 4: Data storage size with different row group sizes in RCFile.

We can see that a large row group size can certainly improve the data compression efficiency and decrease the storage size. Therefore, we could not use a small row group size with the requirement of reducing storage space. However, as shown in Figure 4, when the row group size are larger than 4MB, the compressed data size in the storage almost becomes a constant.

2) The relationship between row group size and lazy decompression: Though a large row group size helps decrease the storage size for a table, it may hurt the data read performance, since a small row group size is a better choice to gain the performance advantage of lazy decompression than a large row group size.

We present an example to demonstrate the relationship between row group size and lazy decompression. Consider a query "*SELECT a,b,c,d FROM tbl WHERE e = 1*". We assume that in an HDFS block, only one record of the table can satisfy the condition of $e = 1$. We now consider an extreme case where the row group size is as large as possible so that the HDFS block contains only one row group. In this case, RCFile will decompress column e first to check the where condition, and find that the condition can be satisfied in this row group. Therefore, RCFile will also decompress all the other four columns (a, b, c, d) since they are needed by the query.

However, since only one record in the HDFS block is actually useful, i.e., the one with $e = 1$, it is not necessary to decompress the other data items of columns (a, b, c, d) except this record. If we use a small row group size, then there is only one useful row group that contains the record with $e = 1$, and all other row groups are useless for this query. Therefore, only in the useful row group, all the five columns (a, b, c, d, e) need to be decompressed. In other useless row groups, only column e needs to be decompressed, and the other four columns will not be decompressed, since the where condition cannot be satisfied. Thus, the query execution time can be decreased.

Users should choose the row group size to consider both the storage space and query execution requirements. Currently, RCFile adapted in Facebook uses 4MB as the default row group size. The RCFile library provides a parameter to allow users to select a size for their own table.

IV. The API of RCFile

RCFile can be used in two ways. First, it can be used as the input or output of a MapReduce program, since it implements the standard interfaces of `InputFormat`, `RecordReader` and `OutputFormat`. This is the way how RCFile is used in Hive. Second, it can also be directly accessed through its interfaces exposed by `RCFile.Reader` and `RCFile.Writer`.

A. Usage in a MapReduce Program

The implementations of `InputFormat`, `RecordReader` and `OutputFormat` in RCFile are `RCFileInputFormat`, `RCFileRecordReader`, and `RCFileOutputFormat`, respectively. When using RCFile in a MapReduce program via these interfaces, there are two major configurations.

First, when using RCFile to read a table, `RCFileRecordReader` needs to know which columns of the table are needed. The following code example demonstrates how to set such information. After the setting, the configuration of *hive.io.file.readcolumn.ids* will be *(2,3)*.

```
ArrayList<Integer> readCols = new ArrayList<Integer>();
readCols.add(Integer.valueOf(2));
readCols.add(Integer.valueOf(3));
ColumnProjectionUtils.setReadColumnIDs(conf, readCols);
```

Second, when using RCFile as output, the number of columns needs to be specified in `RCFileOutputFormat`. The following code example shows the methods to set and get the number of columns. The call of `RCFileOutputFormat.setColumnNumber` will set the configuration value of *hive.io.rcfile.column.number.conf*. `RCFileOutputFormat.getColumnNumber` can be used to get the number of columns.

```
RCFileOutputFormat.setColumnNumber(conf,8);
RCFileOutputFormat.getColumnNumber(conf);
```

B. RCFile Reader and Writer

RCFile also provides Reader and Writer to allow applications to use RCFile in their own ways. RCFile Writer is simple since it only supports data appending.

The interfaces of RCFile Reader are categorized into three sets: common utilities, row-wise reading, and column-wise reading. Table I summarizes these three categories. The following code example outlines how to use row-wise reading methods of RCFile. Lines 1 to 6 define several critical variables. Lines 8 to 9 show the method to set needed columns. In this example, columns 2 and 3 are used. Line 10 uses method `setReadColumnIDs` to pass the information of needed columns to RCFile. A RCFile Reader is created at line 12, and then the while loop (lines 14-15) is used to read all the records one by one via the `getCurrentRow` method.

```
1  ArrayList<Integer> readCols;
2  LongWritable rowID;
3  BytesRefArrayWritable cols;
4  FileSystem fs;
5  Path file;
6  Configuration conf;
7
8  readCols.add(2);
9  readCols.add(3);
10 setReadColumnIDs(conf, readCols);
11
12 Reader reader = new Reader(fs, file, conf);
13
14 while (reader.next(rowID))
15   reader.getCurrentRow(cols);
16
17 reader.close();
```

Notice that calling `getCurrentRow` does not mean that all the fields in the row have been decompressed. Actually, according to lazy decompression, a column will not be decompressed until one of its field is being deserialized. The following code example shows the deserialization of a field in column `i`. If this column has not been decompressed. This procedure will trigger lazy decompression. In the code, variable `standardWritableData` is the deserialized field.

```
row = serDe.deserialize(cols);
oi = serDe.getObjectInspector();
fieldRefs = oi.getAllStructFieldRefs();
fieldData = oi.getStructFieldData(row, fieldRefs.get(i));
standardWritableData =
        ObjectInspectorUtils.copyToStandardObject(
                fieldData,
                fieldRefs.get(i).getFieldObjectInspector(),
                ObjectInspectorCopyOption.WRITABLE);
```

TABLE I: Interfaces of RCFile.Reader

(a) Common Utilities

Method	Action
Reader()	Create RCFile reader
getPosition()	Get the current byte offset from the beginning
close()	Close the reader

(b) Row-wise reading

Method	Action
next()	Return if there is more data available. If yes, advance the current reader pointer to the next record
getCurrentRow()	Return the current row

(c) Column-wise reading

Method	Action
getColumn()	Fetch an array of a specific column data of the current row group
nextColumnsBatch()	Return if there are more row groups available. If yes, discard all bytes of the current row group and advance the current reader pointer to the next row group

V. Performance Evaluation

We have conducted experiments with RCFile and Hive at Facebook to demonstrate the effectiveness of RCFile. The experiments were done in a Facebook cluster with 40 nodes. Most nodes are equipped with an Intel Xeon CPU with 8 cores, 32GB main memory, and 12 1TB disks. The operating system is Linux (kernel version 2.6). In our experiments, we have used the Hadoop 0.20.1, and the most recent version of Hive.

We have conducted two groups of experiments. In the first group, we have compared RCFile with row-store, column-store, and column-group. In the second group, we have evaluated the performance of RCFile from the perspectives of both storage space and query execution, by using different row group sizes and different workloads.

A. RCFile versus Other Structures

In this group of experiments, we configured Hive to use several different data placement structures. Row-store is implemented as shown in Figure 1. For both column-store and column-group, we have used the Zebra library [20], which is widely used in the Pig system [6]. Zebra implements both column-store and column-group. To conduct the experiments, we have integrated the Zebra library into Hive with necessary modifications.

Our goal in this group of experiments is to demonstrate the effectiveness of RCFile in three aspects:

1) data storage space,
2) data loading time, and
3) query execution time.

Our workload is from the the benchmark proposed by Palvo et al. [22]. This benchmark has been widely-used to evaluate different large-scale data processing platforms [23].

1) Storage Space: We selected the *USERVISITS* table from the benchmark, and generated the data set whose size is about 120GB. The generated data is all in plain text, and we loaded it into Hive using different data placement structures. During loading, data is compressed by the Gzip algorithm for each structure.

The table has nine columns: *sourceIP*, *destURL*, *visitDate*, *adRevenue*, *userAgent*, *countryCode*, *languageCode*, *searchWord*, and *duration*. Most of them are of string type. By column-group in Zebra, *sourceIP* and *adRevenue* are located in the same column group, in order to optimize the queries in the benchmark.

Figure 5 shows the storage space sizes required by the raw data and by several data placement structures. We can see that data compression can significantly reduce the storage space, and different data placement structures show different compression efficiencies as follows.

- Row-store has the worst compression efficiency compared with column-store, column-group, and RCFile. This is expected because that a column-wise data compression is better than a row-wise data compression with mixed data domains.

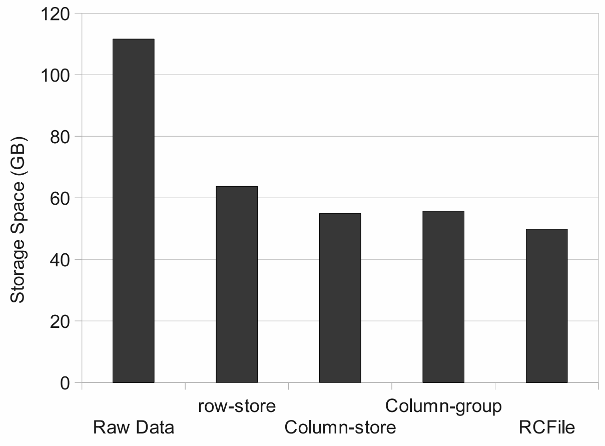


Fig. 5: Sizes (GB) of raw data and loaded tables with different data placement structures in Hive.

- RCFile can reduce even more space than column-store does. This is because the implementation of Zebra stores column metadata and real column data together, so that it cannot compress them separately. Recall the RCFile data layout in Section III-A, 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 can have better data compression efficiency than that of Zebra.

2) Data Loading Time: For a data placement structure, data loading time (the time required by loading the raw data into the data warehouse) is an important factor for daily operations in Facebook. Reducing this time is critical since Facebook has to load about 20TB data into the production warehouse everyday.

We have recorded the data loading times for the USERVISITS table in the above experiment. The results are shown in Figure 6. We have the following observations:

- Among all cases, row-store always has the smallest data loading time. This is because it has the minimum overhead to re-organize records in the raw text file.
- Column-store and column-group have significantly long loading times than both row-store and RCFile. This is because each record in the raw data file will be written to multiple HDFS blocks for different columns (or column groups). Since these multiple blocks are not in the same cluster node, data loading will cause much more network overhead than using other structures.
- RCFile is slightly slower than row-store with a comparable performance in practice. This reflects the small overhead of RCFile since it only needs to re-organize records inside each row group whose size is significantly smaller than the file size.

3) Query execution time: In this experiment, we executed two queries on the *RANKING* table from the benchmark. The table has three columns: *pageRank*, *pageURL*, and *avgDuration*. Both of column *pageRank* and *avgDuration* are of integer type, and *pageURL* is of string type. In Zebra, when using column-group, we organized *pageRank* and *pageURL* in the

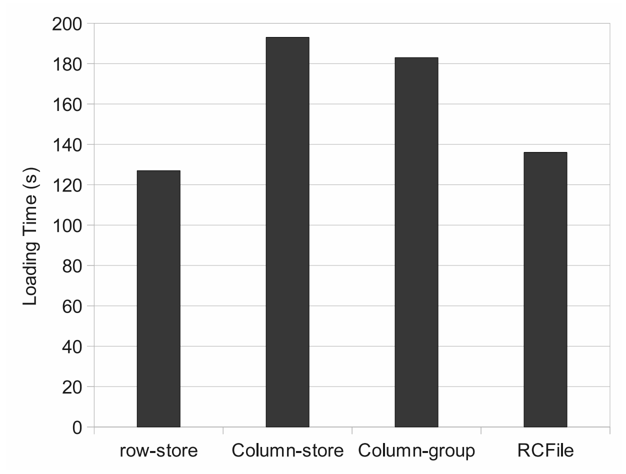


Fig. 6: Data loading times.

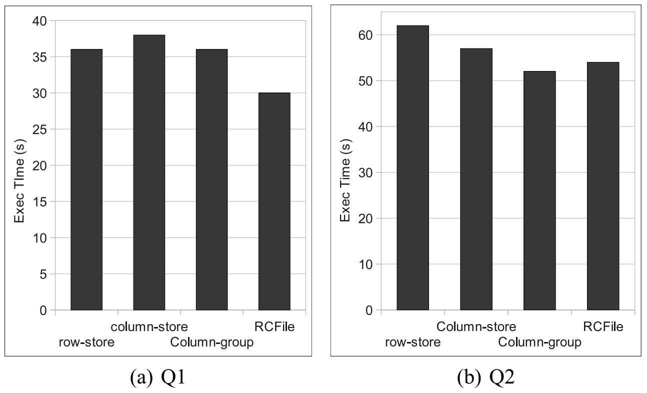


Fig. 7: Query execution times.

same column group. When using column-store, each of the three columns is stored independently. The queries are shown as follows.

```
Q1: SELECT pagerank, pageurl FROM RANKING
    WHERE pagerank > 400;

Q2: SELECT pagerank, pageurl FROM RANKING
    WHERE pagerank < 400;
```

In order to evaluate the performance of lazy decompression of RCFile, the two queries were designed to have different selectivities according to their where conditions. It is about 5% for ($pagerank > 400$), and about 95% for ($pagerank < 400$).

Figure 7a and 7b show the execution times of the two queries with the four data placement structures. We have two major observations.

- For Q1, RCFile outperforms the other three structures significantly. This is because the lazy decompression technique in RCFile can accelerate the query execution with a low query selectivity.
- For Q2, column-group has the fastest query execution speed since the high selectivity of this query makes lazy decompression useless in this case. However, RCFile still outperforms column-store. Note that the performance advantage of column-group is not free. It highly relies on pre-defined column combinations before query executions.

4) Summary: We have compared RCFile with other data placement structures in three aspects of storage space, data loading time and query execution time. We show that each structure has its own merits for only one aspect. In contrast, our solution RCFile, which adopts advantages of other structures, is the best choice in almost all the cases.

B. RCFile with Different Row Group Sizes

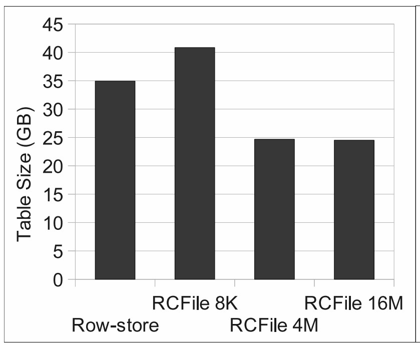
In this group of experiments, we have examined how the row group size can affect data storage space and query execution time. We used two different workloads. The first workload is the industry standard TPC-H benchmark for warehousing system evaluations. The second workload is generated by daily operations for advertisement business in Facebook.

1) TPC-H workload: We selected Q1 and Q6 from the benchmark as our queries. Both the two queries execute aggregations on the largest table *LINEITEM*. In our experiment, we examined the storage space for this table and query execution times with three different row group sizes (8KB, the default 4MB, and 16MB). The *LINEITEM* has 16 columns, and the two queries only use a small number of columns, and never touch the largest column *l_comments*. In addition, we have also tested row-store structure for performance comparisons.

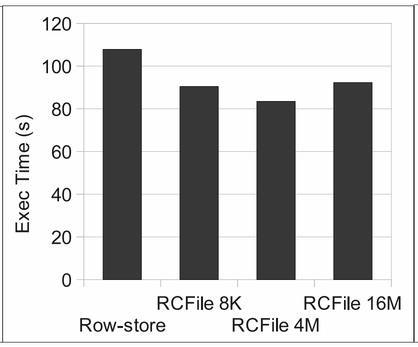
Figure 8 (a) shows the storage space for different configurations. We can see that with a small 8KB row group size, RCFile needs more storage space than row-store. However, when increasing the row group size to 4MB and 16MB, RCFile can significantly decrease storage space compared with row-store. In addition, in this Figure, we cannot see a big gap between the cases of 4MB and 16MB. This means that increasing row group size after a threshold would not help improve data compression efficiency significantly.

Figure 8 (b) and (c) show the query execution times of Q1 and Q6, respectively. We can see that RCFile, no matter with what row group size, can significantly outperform row-store. This reflects the advantage of RCFile over row-store, which skips unnecessary columns as column-store structure does.

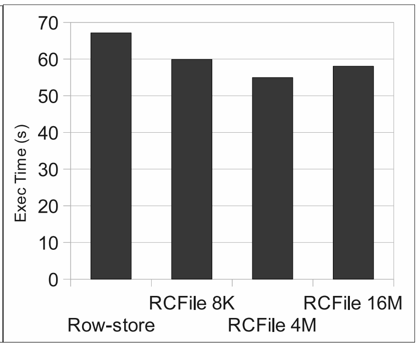
Among the three row group sizes, the middle value (4MB) achieves the best performance. As shown in Figure 8 (a), since the 4MB row group size can decrease table size much more effectively than a small size of 8KB, it can significantly reduce the amount of disk accesses and thus accelerate query execution. It is also not surprisingly to find that the large 16MB row group size is slower than 4MB for both queries. First, as shown in Figure 8 (a), the two sizes (16MB and 4MB) have almost the same compression efficiency. Thus, the large row group size cannot gain further I/O space advantage. Second, according to the current RCFile's implementation, it has more overhead to manage a large row group that must be decompressed and held in memory. Furthermore, a large row group can also decrease the advantage of lazy



(a) Table Size

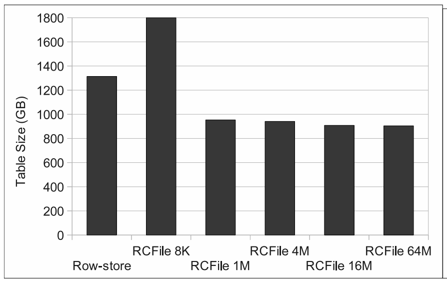


(b) TPC-H Q1

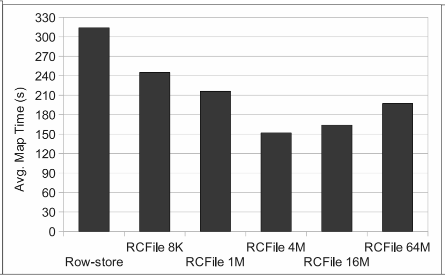


(c) TPC-H Q6

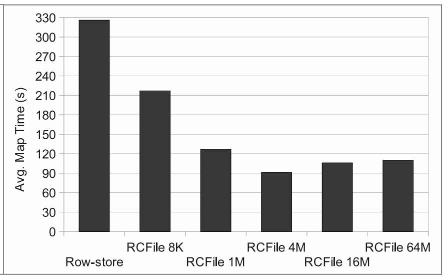
Fig. 8: The TPC-H workloads.



(a) Table Size



(b) Query A



(c) Query B

Fig. 9: The Facebook workload.

decompression, and cause unnecessary data decompression as we have discussed in Section III-D.

2) Facebook workload: We have further used a Facebook workload to examine how the row group size can affect RCFile's performance. In this experiment, we examined different row group sizes (8KB, 1MB, 4MB, 16MB, and 64MB), The experiment was conducted on the table to store the advertisement click-stream in Facebook. The table (namely *adclicks*) is very wide with 38 columns. In this experiment, we used the trace collected in one day, and its size by row-store with compression is about 1.3TB. We used two queries as follows.

```
Query A: SELECT adid, userid FROM adclicks;

Query B: SELECT adid, userid FROM adclicks
         WHERE userid="X";
```

Figure 9 (a) shows the table sizes. Figure 9 (b) and (c) show the average mapper time of the MapReduce job for the query execution of Query A and Query B. The mapper time reflects the performance of the underlying RCFile structure. From the viewpoints of both data compression and query execution times, these results are consistent with the previously presented TPC-H results. In summary, all these results show that a range of from 4MB (the default value) to 16MB is an effective selection as the row group size.

In addition, when comparing the query execution times of Query A and Query B in Figure 9 (b) and (c), we can have two distinct observations. First, for row-store, the average mapper time of Query B is even longer than that of Query A. This is expected since Query B has a where condition that causes more computations. Second, however, for RCFile with the row group size, the average mapper time of Query B is significantly shorter than that of Query A. This reflects the performance benefit of lazy decompression of RCFile. Due to the existence of a where condition in Query B, a lazy decompression can avoid to decompress unnecessary data and thus improve query execution performance.

VI. Other Related Work

We have introduced and evaluated the row-store, the column-store/column-group, and the hybrid PAX store implemented in conventional database systems in Section II. Detailed comparison, analysis, and improvement of these three structures in data compression and query performance can be found in [17][15][24][18][12][11].

In the context of MapReduce-based data warehouse systems, Zebra [20] was the first effort to utilize the column-store/column-group in the Yahoo Pig system on top of Hadoop. As we have discussed in Section II and evaluated in Section V, Zebra's performance is highly dependent on how column groups have been pre-defined. The most recently related work to RCFile is a data storage component in the *Cheetah* system [25]. Like RCFile, the Cheetah system also first horizontally partitions a table into small units (each unit is called a *cell*), and then vertically stores and compresses each column independently. However, Cheetah will further use Gzip to compress the whole cell. Thus, during query execution, Cheetah has to read from the storage and decompress the whole cell before processing any data in a cell. Compared to Cheetah, RCFile can skip unnecessary column reads by independent column compression, and avoid unnecessary column decompression by the lazy decompression technique.

Other related work includes the Google Bigtable system [26] and its open-source implementation Hbase [27] built on Hadoop. The major difference between RCFile and Bigtable/Hbase is that RCFile serves as a storage structure for the almost read-only data warehouse system, while Bigtable/Hbase is mainly a low-level key-value store for both read- and write-intensive applications.

VII. Conclusion

A fast and space-efficient data placement structure is very important to big data analytics in large-scale distributed systems. According to our analysis on Facebook production systems, big data analytics in a MapReduce-based data warehouse has four critical requirements to the design and implementation of a data placement structure, namely 1) fast data loading, 2) fast query processing, 3) highly efficient storage space utilization, and 4) strong adaptivity to highly dynamic workload patterns. Our solution RCFile is designed to meet all the four goals, and has been implemented on top of Hadoop. First, RCFile has comparable data loading speed and workload adaptivity with the row-store. Second, RCFile is read-optimized by avoiding unnecessary column reads during table scans. It outperforms the other structures in most of cases. Third, RCFile uses column-wise compression and thus provides efficient storage space utilization.

RCFile has been integrated into Hive, and plays an important role in daily Facebook operations. It is now used as the default storage structure option for internal data processing systems at Facebook. In order to improve storage space utilization, all the recent data generated by various Facebook applications since 2010 have been stored in the RCFile structure, and the Facebook data infrastructure team is currently working to transform existing data sets stored in the row-store structure at Facebook into the RCFile format.

RCFile has been adopted in data processing systems beyond the scope of Facebook. An integration of RCFile to Pig is being developed by Yahoo! RCFile is used in another Hadoop-based data management system called Howl (http://wiki.apache.org/pig/Howl). In addition, according to communications in the Hive development community, RCFile has been successfully integrated into other MapReduce-based data analytics platforms. We believe that RCFile will continue to play its important role as a data placement standard for big data analytics in the MapReduce environment.

VIII. Acknowledgments

We thank Joydeep Sen Sarma for his initial thought about RCFile. We appreciate Bill Bynum for his careful reading of the paper. We thank anonymous reviewers for their feedback to improve the readability of the paper. This work is supported in part by China Basic Research Program (2011CB302500, 2011CB302800), the Co-building Program of Beijing Municipal Education Commission, and the US National Science Foundation under grants CCF072380 and CCF0913050.

References

[1] http://www.facebook.com/press/info.php?statistics.
[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.
[3] http://hadoop.apache.org/.
[4] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive - a warehousing solution over a Map-Reduce framework," *PVLDB*, vol. 2, no. 2, pp. 1626–1629, 2009.
[5] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy, "Hive - a petabyte scale data warehouse using hadoop," in *ICDE*, 2010, pp. 996–1005.
[6] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a highlevel dataflow system on top of MapReduce: The Pig experience," *PVLDB*, vol. 2, no. 2, pp. 1414–1425, 2009.
[7] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. S. Sarma, R. Murthy, and H. Liu, "Data warehousing and analytics infrastructure at facebook," in *SIGMOD Conference*, 2010, pp. 1013–1020.
[8] R. Ramakrishnan and J. Gehrke, "Database management systems," McGraw-Hill, 2003.
[9] G. P. Copeland and S. Khoshafian, "A decomposition storage model," in *SIGMOD Conference*, 1985, pp. 268–279.
[10] 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*, 2005, pp. 553–564.
[11] D. J. Abadi, S. Madden, and N. Hachem, "Column-stores vs. row-stores: how different are they really?" in *SIGMOD Conference*, 2008.
[12] A. L. Holloway and D. J. DeWitt, "Read-optimized databases, in depth," *PVLDB*, vol. 1, no. 1, pp. 502–513, 2008.
[13] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving relations for cache performance," in *VLDB*, 2001, pp. 169–180.
[14] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe, "Query processing techniques for solid state drives," in *SIGMOD Conference*, 2009, pp. 59–72.
[15] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden, "Performance tradeoffs in read-optimized databases," in *VLDB*, 2006, pp. 487–498.
[16] M. Stonebraker and U. Çetintemel, "One size fits all: An idea whose time has come and gone (abstract)," in *ICDE*, 2005, pp. 2–11.
[17] V. Raman and G. Swart, "How to wring a table dry: Entropy compression of relations and querying of compressed relations," in *VLDB*, 2006.
[18] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle, "Constant-time query processing," in *ICDE*, 2008, pp. 60–69.
[19] P. A. Boncz, S. Manegold, and M. L. Kersten, "Database architecture optimized for the new bottleneck: Memory access," in *VLDB*, 1999.
[20] http://wiki.apache.org/pig/zebra.
[21] R. A. Hankins and J. M. Patel, "Data morphing: An adaptive, cache-conscious storage technique," in *VLDB*, 2003, pp. 417–428.
[22] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *SIGMOD Conference*, 2009, pp. 165–178.
[23] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz, "HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads," *PVLDB*, vol. 2, no. 1, pp. 922–933, 2009.
[24] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt, "How to barter bits for chronons: compression and bandwidth trade offs for database scans," in *SIGMOD Conference*, 2007, pp. 389–400.
[25] S. Chen, "Cheetah: A high performance, custom data warehouse on top of mapreduce," *PVLDB*, vol. 3, no. 2, pp. 1459–1468, 2010.
[26] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," in *OSDI*, 2006, pp. 205–218.
[27] http://hbase.apache.org.