# Understanding Insights into the Basic Structure and Essential Issues of Table Placement Methods in Clusters

Yin Huai[1] Siyuan Ma[1] Rubao Lee[1] Owen O'Malley[2] Xiaodong Zhang[1]

[1]Department of Computer Science and Engineering, The Ohio State University
[2]Hortonworks Inc.

[1]{huai, masi, liru, zhang}@cse.ohio-state.edu [2]owen@hortonworks.com

## ABSTRACT

A table placement method is a critical component in big data analytics on distributed systems. It determines the way how data values in a two-dimensional table are organized and stored in the underlying cluster. Based on Hadoop computing environments, several table placement methods have been proposed and implemented. However, a comprehensive and systematic study to understand, to compare, and to evaluate different table placement methods has not been done. Thus, it is highly desirable to gain important insights into the basic structure and essential issues of table placement methods in the context of big data processing infrastructures.

In this paper, we present such a study. The basic structure of a data placement method consists of three core operations: row reordering, table partitioning, and data packing. All the existing placement methods are formed by these core operations with variations made by the three key factors: (1) the size of a horizontal logical subset of a table (or the size of a row group), (2) the function of mapping columns to column groups, and (3) the function of packing columns or column groups in a row group into physical blocks. We have designed and implemented a benchmarking tool to provide insights into how variations of each factor affect the I/O performance of reading data of a table stored by a table placement method. Based on our results, we give suggested actions to optimize table reading performance. Results from large-scale experiments have also confirmed that our findings are valid for production workloads. Finally, we present ORC File as a case study to show the effectiveness of our findings and suggested actions.

## 1. INTRODUCTION

Structured data analytics is a major Hadoop application, in the context of which various query execution systems and query translators have been designed and implemented, such as Hive [1], Pig [2], Impala [3], Cheetah [19], Shark [27], and YSmart [24]. In these systems and translators, a critical component is a table placement method that determines how the data values in a two-dimensional table are organized and stored on the storage devices. One critical issue of a table placement method is the I/O performance of reading data from the placed table since it can fundamentally affect the overall query execution performance.

In Hadoop-based computing environments, several table placement methods have been proposed and implemented, such as CFile in Llama [25], Column Format (CF) [20], Record Columnar File (RCFile) [22] in Hive [1], Optimized Record Columnar File (ORC File) [4] in Hive, Parquet [5], Segment-Level Column Group Store (SLC-Store) in Mastiff [21], Trevni [6] in Avro [7], and Trojan Data Layouts (TDL) [23]. These projects have been done independently, and a comprehensive and systematic study of table placement methods is absent, leaving three critical issues unaddressed.

1. The basic structure of a table placement method has not been defined. This structure should abstract the core operations to organize and to store data values of a table in the underlying cluster. It also serves as a foundation to design and implement a table placement method.
2. A fair comparison among different table placement methods is almost impossible due to heavy influences of diverse system implementations, various auxiliary data and optimization techniques, and different workload configurations. Performance evaluations from existing work commonly report overall experimental results. Influences from different sources are hard to be distinguished.
3. There is no general guideline on how to adjust table placement methods to achieve optimized I/O read performance. To adapt various workload patterns and different underlying systems, a systematic approach to optimize a table placement method is highly desirable.

In this paper, we present our study aiming to address the above three issues. Our study focuses on the basic structure of a table placement method, which describes core operations to organize and store data values of a table. With the basic structure, we define different table placement methods and identify critical factors on which these methods are different from each other. Then, we comprehensively and systematically evaluate impacts of these factors on the aspect of I/O read performance by our micro-benchmarks. Based on the experimental results of our micro-benchmarks, we provide general guidelines for performance optimization. To show impacts of different table placement methods on production workloads, we also provide experimental results of several large-scale experiments (macro-benchmarks). These results confirm our findings from micro-benchmarks. Then, we discuss the trade-off between the data reading efficiency and the degree of parallelism when choosing a suitable row group size. Finally, we present ORC File as a case study to show the effectiveness of our findings and suggested actions.

*Proceedings of the VLDB Endowment, Vol. 6, No. 14*
*Copyright 2013 VLDB Endowment 2150-8097/13/14... $ 10.00.*

Our study makes the following three main contributions.

1. We define the basic structure of a table placement method, which is used to form a table placement method, to abstract existing implementations, and to characterize differences between existing table placement methods.
2. We design and implement a benchmarking tool (for micro-benchmarks) to experimentally study different design factors of table placement methods. To provide a fair evaluation and insights into essential issues, this benchmarking tool uses an implementation of a table placement method (also called a table-placement-method implementation) to simulate variations of each design factor.
3. We comprehensively study data reading issues related to table placement methods and provide guidelines on how to adjust table placement methods to achieve optimized I/O read performance. Our guidelines are applicable to different table placement methods. Because I/O read performance is closely related to table placement methods and is a critical factor of the overall data processing performance, we believe that our results and guidelines lay a foundation to existing and future table placement methods.

The remainder of this paper is organized as follows. Section 2 presents the basic structure of table placement methods and describes existing table placement methods under this structure. Section 3 gives an overview of our study methodology. Section 4 details our experimental results of micro-benchmarks. Section 5 details our experimental results of macro-benchmarks. Section 6 discusses the trade-off when choosing a suitable row group size in clusters. Section 7 introduces ORC File in which we explain its design as a case study. Section 8 is the conclusion.

## 2. TABLE PLACEMENT METHOD

In this section, we first provide the definition of the basic structure of table placement methods. Through this definition, we are able to study existing table placement methods under a unified way. Then, we summarize design factors on which existing table placement methods are different from each other. Finally, we describe how to use implementations of RCFile and Trevni to simulate variations of each design factor.

## 2.1 Definition

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*. 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. We will use the position of a specific row in the row sequence to refer to this row, e.g. the first row. Also, all columns of a table form a column sequence. We will use the position of a specific column in the column sequence to refer to this column, e.g. the second column. In this way, we use the position to refer to a specific data value in the table, e.g. the data value at the first row and the second column.

### *2.1.1 Row Reordering*

The row reordering procedure rearranges rows of a given table based on a given function $f_{RR}$ shown in Equation 1.

$$i' = f_{RR}(i). \tag{1}$$

This function will form a new sequence of rows by assigning a new position $i'$ to a row referred to as $i$ in the original sequence of rows. For example, $f_{RR}(1) = 10$ means that the first row in the original

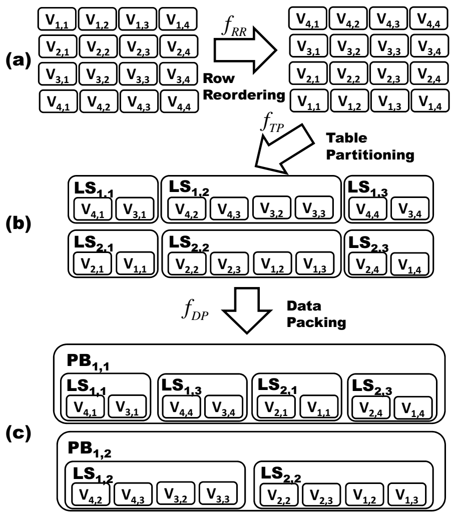


**Figure 1: A demonstration of a table placement method.** $V_{i,j}$ **represents the data value at the** $i$**th row and** $j$**th column. For the purpose of a clear presentation, all two dimensional indexes of a data value are generated based on the original table.**

table will be the 10th row after the row reordering procedure. It is worth noting that rows in the table will not be reordered in the subsequent two procedures of table partitioning and data packing.

There are two representative examples of the row reordering procedure. First, the entire table is reordered based on the data values of certain columns. Second, we can divide the table to multiple non-overlapping subsets and then reorder every subset. This row reordering procedure can be used when reordering the entire table is not cost-effective or not feasible. One representative purpose of the row reordering procedure is to encode or compress data values efficiently. A demonstration of a simple row reordering procedure is shown in Figure 1(a). In this figure, the row reordering procedure reverses the original row sequence.

### *2.1.2 Table Partitioning*

In general, the table partitioning procedure divides the entire set of data values of the table into multiple non-overlapping subsets (called *logical subsets*) based on a given partitioning function $f_{TP}$ shown in Equation 2.

$$LS_{x,y} = f_{TP}(i, j). \tag{2}$$

Through $f_{TP}$, the value at the $i$th row and the $j$th column [1] is assigned to the logical subset $LS_{x,y}$. Like a value in a table, a logical subset $LS_{x,y}$ is identified by a two-dimensional index, which is $(x, y)$, and we also refer to this logical subset as the one located at the $x$th *logical-subset-row* and the $y$th *logical-subset-column*. For example, in Figure 1(b), the original table is divided to 6 logical subsets, where $LS_{1,1}$ belongs to the first logical-subset-row and the first logical-subset-column. In a logical subset, values will be stored in a row-by-row fashion, i.e. a value at the $i_1$th row and the $j_1$th column is stored before the one at the $i_2$th row and the $j_2$th if and only if $i_1 < i_2$, or $i_1 = i_2$ and $j_1 < j_2$.

Table partitioning functions in existing table placement methods (Section 2.2) commonly exhibit two properties which are

1. $\forall i, \forall j_1, \forall j_2$, when $LS_{x_1,y_1} = f_{TP}(i, j_1)$ and $LS_{x_2,y_2} = f_{TP}(i, j_2)$, we have $x_1 = x_2$; and

[1]To be specific, the value is at the $i$the row and $j$the column of the reordered table, i.e. the output of the row reordering procedure.

Table 1: A summary of symbols used in Table 2

| Symbol | Meaning |
|---|---|
| $C$ | a set of columns |
| $C_i$ | the $i$th column |
| $C_{sort}$ | a subset of columns used as sorting keys |
| $f_{CG}(C_i)$ | the column group presented by an integer index that the column $C_i$ belongs to. For example, if there are $n$ column groups, the range of $f_{CG}$ is $[1, n]$. |
| $RG$ | a set of row groups (logical-subset-rows). |
| $RG_i$ | the $i$th row group. |
| $S_r(i)$ | the size of the $i$th row |
| $S(LS_{i,j})$ | the size of the logical subset $LS_{i,j}$ |
| $S_{PB}$ | the user-specified maximum size of a physical block |
| $S(RG_i, C_j)$ | the size of the column $C_j$ in the row group $RG_i$ |

2. $\forall i_1, \forall i_2, \forall j$, when $LS_{x_1,y_1} = f_{TP}(i_1, j)$ and $LS_{x_2,y_2} = f_{TP}(i_2, j)$, if $i_1 < i_2$, then $x_1 \leq x_2$.

For these table partitioning functions, a logical-subset-row represents a set of contiguous rows. In this case, a *row group* is also used to refer to a logical-subset-row.

### 2.1.3 Data Packing

After a table being divided into multiple logical subsets, the data packing procedure will place those logical subsets into *physical blocks* based on a given function $f_{DP}$ shown in Equation 3.

$$PB_{p,q} = f_{DP}(LS_{x,y}). \quad (3)$$

Through this function, a logical subset $LS_{x,y}$ will be assigned to a physical block $PB_{p,q}$, which is identified by a two-dimensional index, $(p, q)$. For example, in Figure 1(c), logical subsets $LS_{1,1}$, $LS_{1,3}$, $LS_{2,1}$, and $LS_{2,3}$ are packed into the physical block $PB_{1,1}$. A physical block is filled by a set of logical subsets and two different physical blocks do not have any common logical subset. Also, a physical block is the storage unit of the underlying storage system. For example, a HDFS block is a physical block in Hadoop distributed filesystem (HDFS). In a physical block, logical subsets are stored by the order of their indexes, i.e. $LS_{x_1,y_1}$ is stored before $LS_{x_2,y_2}$ if and only if $x_1 < x_2$, or $x_1 = x_2$ and $y_1 < y_2$.

## 2.2 Existing Work

With the basic structure of a table placement method defined in Section 2.1, we are able to describe different table placement methods in a unified way. Here, we choose Column Format (CF), CFile, Record Columnar File (RCFile), Segment-Level Column Group Store (SLC-Store), Trevni, and Trojan Data Layouts (TDL) as six examples to show the generality of our definition of the basic structure. For the purpose of simplicity, we consider that a column has a primitive data type (e.g. `Integer`, `Double` and `String`). If a column has a complex data type (e.g. `Map`), it can be considered as a single column or it can be decomposed to multiple columns with primitive types.

Optimized Record Columnar File (ORC File) and Parquet are not presented at here due to space limit. For primitive data types, they are similar to RCFile from the perspective of the basic structure. For complex data types, they provide their own approaches to decompose a column with a complex data type to multiple columns with primitive data types. After the decomposition, a table can be considered as one with only primitive data types. Other features of these two data placement methods are beyond the scope of this paper. Interested readers may refer to [8] for ORC File and to [5] for Parquet.

Using the symbols summarized in Table 1, we present the definitions of these six table placement methods in Table 2.

### 2.2.1 Row Reordering Variations

Among those six table placement methods in Table 2, CFile and SLC-Store can specify the functions of the row reordering procedure. CFile applies a sort to the entire table based on certain columns. SLC-Store can apply an optional sort to rows in every in-memory write buffer based on certain columns before flushing data to disks. These specific row reordering procedures target specific workloads. For example, CFile is designed for join operations [25]. Other table placement methods do not specify the row reordering function, and different functions can be applied as a pre-process.

### 2.2.2 Table Partitioning Variations

Regarding the procedure of table partitioning, those six table placement methods are divided into three categories. In the first category, CF, RCFile, and Trevni only store a single column in a row group to a logical subset. Also, these three methods set that the sizes of all row groups are the same. CFile is in the second category. Like methods in the first category, it stores a single column in a row group to a logical subset. However, CFile sets that all row groups have the same number of rows. In the third category, SLC-Store and TDL both store a column group (a subset of columns) [2] in a row group to a logical subset, and the sizes of all row groups are the same. For these two table placement methods, a function mapping columns to column groups is required.

### 2.2.3 Data Packing Variations

On the data packing procedure, those six table placement methods are divided into four categories. In the first category, CF stores a single logical subset to a physical block. In the second category, CFile stores multiple logical subsets of a logical-subset-column to a physical block. In the third category, RCFile stores multiple row groups to a physical block. In the fourth category, SLC-Store, Trevni [3], and TDL store a single row group to a physical block.

### 2.2.4 Summary

Comparing six existing representative table placement methods, we find that they are different on four design factors, which determine their differences on functions of $f_{RR}$, $f_{TP}$, and $f_{DP}$. These factors are:

1. **Row reordering:** How to reorder rows in the table. Variations in this factor aim to facilitate specific workloads.
2. **Row group size:** The size of a row group.
3. **Grouping columns:** How to group columns.
4. **Packing column groups:** How to pack column groups in a row group into physical blocks. This factor is dependent on underlying distributed filesystems.

The first factor determines the row reordering function $f_{RR}$. Then, the second and third factors determine the table partitioning function $f_{TP}$. Finally, the fourth factor determines the data packing function $f_{DP}$.

## 2.3 A Unified Evaluation Framework

Although table placement methods shown in Table 2 are different from each other, to provide a fair evaluation, we use a single implementation to simulate variations of each design factor of table

[2]A column group can have only one column.

[3]Although the implementation of Trevni allows that a Trevni file can be larger than a physical block, according to the design goal of Trevni described in [6], we consider that the size of a file of Trevni is equal to the size of a physical block.

Table 2: **Definitions of six table placement methods. For the column of $f_{RR}$, NS means not specified, and *sort1* and *sort2* are sorting functions specified in corresponding table placement methods.**

| Name | $f_{RR}$ | $f_{TP}(i,j)$ | $f_{DP}(x,y)$ | Restrictions |
|---|---|---|---|---|
| CF | NS | $(\lceil \frac{\sum_{k=1}^{i} S_r(k)}{S_{RG}} \rceil, j)$ | $(x,y)$ | 1. all row groups have the same size $S_{RG}$ <br> 2. $\max(\{S(RG_i, C_j) \| RG_i \in RG, C_j \in C\}) \leq S_{PB}$ |
| CFile | *sort1*$(C_{sort})$ | $(\lceil \frac{i}{N_{RG}} \rceil, j)$ | $(\lceil \frac{\sum_{k=1}^{x} S(LS_{k,y})}{S_{PB}} \rceil, y)$ | 1. all row groups have the same number of rows $N_{RG}$ <br> 2. $\max(\{S(RG_i, C_j) \| RG_i \in RG, C_j \in C\}) \leq S_{PB}$ |
| RCFile | NS | $(\lceil \frac{\sum_{k=1}^{i} S_r(k)}{S_{RG}} \rceil, j)$ | $(\lceil \frac{S_{RG} \times x}{S_{PB}} \rceil, 1)$ | 1. all row groups have the same size $S_{RG}$ <br> 2. $S_{RG} \leq S_{PB}$ |
| SLC-Store | *sort2*$(C_{sort})$ | $(\lceil \frac{\sum_{k=1}^{i} S_r(k)}{S_{RG}} \rceil, f_{CG}(C_j))$ | $(x,1)$ | 1. all row groups have the same size $S_{RG}$ <br> 2. $S_{RG} = S_{PB}$ |
| Trevni | NS | $(\lceil \frac{\sum_{k=1}^{i} S_r(k)}{S_{RG}} \rceil, j)$ | $(x,1)$ | 1. all row groups have the same size $S_{RG}$ <br> 2. $S_{RG} = S_{PB}$ |
| TDL | NS | $(\lceil \frac{\sum_{k=1}^{i} S_r(k)}{S_{RG}} \rceil, f_{CG}(C_j))$ | $(x,1)$ | 1. all row groups have the same size $S_{RG}$ <br> 2. $S_{RG} = S_{PB}$ |

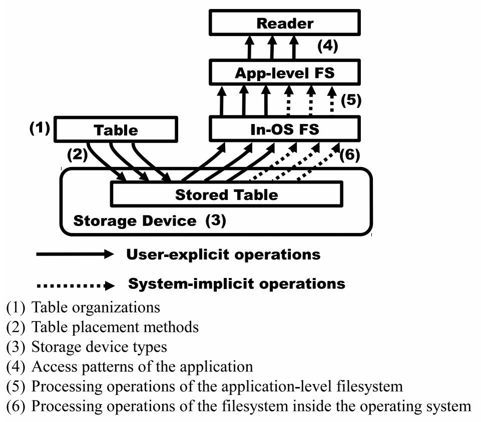

Reader
(4)
App-level FS
(5)
(1)
Table
In-OS FS
(2)
(6)
Stored Table
Storage Device (3)
User-explicit operations
System-implicit operations
(1) Table organizations
(2) Table placement methods
(3) Storage device types
(4) Access patterns of the application
(5) Processing operations of the application-level filesystem
(6) Processing operations of the filesystem inside the operating system

**Figure 2: The entire process of accessing a table at run time.**

placement methods. In this way, we are able to only focus on variations of the basic structure and to eliminate impacts from diverse codebases, different optimizations, and various auxiliary data,

Because only implementations of RCFile and Trevni were open sourced at the point when we started this study, we use these two as examples to illustrate how to simulate variations of each design factor summarized in Section 2.2.4.

**Row reordering:** Row reordering can be done as a pre-process of storing tables to RCFile or Trevni. Different approaches of row reordering can be implemented.

**Row group size:** In RCFile, the size of a row group can be adjusted. Trevni originally stores a single row group in a physical block. We can adjust the row group size by adjusting the size of a physical block. For example, to decrease the row group size used in Trevni-based simulations, applications can explicitly write less number of rows to a physical block before switching to write a new physical block.

**Grouping columns:** In RCFile and Trevni, we can use a composite column (e.g. `STRUCT` in Hive) to store multiple grouped columns in a row by row fashion.

**Packing column groups:** We pack column groups in a row group into one or multiple physical blocks by writing to multiple RCFile or Trevni files.

## 3. STUDY METHODOLOGY

In this work, we study the impacts of different table placement methods on I/O performance of read operations for three reasons. First, since a table placement method describes how data in a table are organized and stored in the underlying system, the results reasoned from the evaluation of I/O read performance are useful to different table-placement-method implementations. Second, computing performance is tightly coupled with the implementation, but a fair comparison among different table placement methods requires eliminating impacts from different implementations. Thus, evaluating computing performance is fundamentally against the purpose of our study. Third, a table placement method aims to store a table in a way through which applications can efficiently access the table. For this reason, the performance of read operations is a major concern for table placement methods.

Next, we identify six factors that affect the I/O performance of reading data from a table. Figure 2 illustrates the entire process of accessing a table. A table is stored in the storage device with a given table placement method. Then, applications (readers) will access the data of this table through an application-level filesystem (e.g. HDFS) and the filesystem inside the local operating system (e.g. ext4 in Linux). In this entire process, there are a total number of six factors (numbered in Figure 2) which can have impacts on the performance of data accessing. These six factors are:

1. Table organizations, which represent data structures of the table, e.g. the number of columns, the type of a column, and the size of a value in a column;
2. Table placement methods, which describe how values in the table are organized and stored in the underlying system;
3. Storage device type, which represents the device actually storing the table, e.g. a HDD or an SSD;
4. Access patterns of the application, which represent how an application accesses the data (user-explicit operations), i.e. a sequence of read operations, each of which starts at a certain position of the file(s) storing the table and loads a certain amount of data from the underlying system;
5. Processing operations of the filesystem at the application level (application-level filesystem), which represent (1) the way that the application-level filesystem issues read operations originally from the application to the filesystem inside the

local OS, and (2) the background read operations (system-implicit operations) issued by the application-level filesystem, i.e. application-level filesystem asynchronous readahead;
6. Processing operations of the filesystem inside the local OS, which represent (1) the way that the local OS issues read operations from the application-level filesystem to the storage device, and (2) the background read operations (system-implicit operations) issued by this local OS, i.e. local-OS asynchronous readahead;

To understand the impact of variations of table placement methods, we need to study how table placement methods are interacted with other five factors listed above.

To provide a comprehensive and unbiased evaluation, we used two sets of benchmarks in our experiments and our experiments were designed under the guidance of the unified evaluation framework described in Section 2.3. First, a set of micro-benchmarks was used to show insights into I/O read performance of a single Map task. We set three requirements for our micro-benchmarks:

1. Different software implementations of table placement methods should not affect the results of the study;
2. Experimental cases used in the study should cover a wide range to provide comprehensive results; and
3. Factors not shown in Figure 2 should be eliminated.

Experimental results from our micro-benchmarks will be presented in Section 4. Second, we used a set of macro-benchmarks to show implications of different table placement methods in real applications. We present results of our macro-benchmarks in Section 5.

## 4. MICRO-BENCHMARK RESULTS

### 4.1 Rationality and Objectives

In an execution environment of Hadoop, reading a table is decomposed into several independent read operations executed by independent Map tasks. Thus, gaining insights into performance of a single Map task is critical to understand and to improve the overall table reading performance. However, gaining insights into a single Map task is hard when benchmarking a table placement method using real-world applications because lots of factors other than table placement methods can affect the performance. To solve this issue, micro-benchmarks are needed. We aim to design a set of micro-benchmarks that can be controlled to individually test each factor summarized in Section 2.2.4. This set of micro-benchmarks should also provide insights which are applicable to different implementations of table placement methods.

### 4.2 Controlled Experimental Environment

In this section, we describe our controlled experimental environment. Specifically, we detail those six factors described in Section 3 that affect the I/O performance of reading data from a table.

#### 4.2.1 *Table Organizations*

Since we focus on I/O read performance, only two aspects of the table organization matter in our study, which are the number of columns and the size of a value of a column. Because impacts from these two aspects are predictable, we believe that showing the results on tables with a fixed number of columns and a fixed value size is able to provide insights that are applicable to different table organizations. Thus, we used tables with 16 columns in our experiments and the data type of each column was string. We fixed the length of a string, which was 40 characters. In our micro-benchmarks, two tables $T_1$ and $T_2$ were used. $T_1$ had 2100000 rows, and the size of a column was 80.1 MiB [4]. While, $T_2$ had 700000 rows, and the size of a column was 26.7 MiB.

#### 4.2.2 *Table Placement Methods*

Since impacts of different row reordering functions ($f_{RR}$) are tightly coupled with workloads and we aim to provide workload-independent insights, we did not use any row reordering function. We only studied impacts of different table partitioning functions ($f_{TP}$) and data packing functions ($f_{DP}$). Specifically, we considered different functions of $f_{TP}$ and $f_{DP}$ based on the second, third, and fourth factors described in Section 2.2.4.

$T_1$ and $T_2$ were stored with different table placement methods. We refer to every stored table by a 3-tuple $(f_{TP}, f_{DP}, \textit{table})$, where $f_{TP}$, $f_{DP}$, and *table* are the table partitioning function, the data packing function, and the name of the table, respectively.

$f_{TP}$ is represented by a 2-tuple $(RG, CG)$, where $RG$ shows the size of a row group, e.g. *64 MiB*, and $CG$ means how columns were grouped. $CG$ can be *grouped*, which means that 2 columns were grouped into a column group in our micro-benchmarks, or it can be *non-grouped*, which means that a single column was stored in a column group. Combining both $RG$ and $CG$, we can know the table partitioning function $f_{TP}$ of a stored table. For example, $(\textit{256 MiB}, \textit{grouped})$ means that the row group size was 256 MiB, and 2 columns were grouped into a column group. Also, we use $*$ to indicate that varied options were used. For example, $(*, \textit{non-grouped})$ means that varied row group sizes were used and a column group only had a single column. If a table was stored by Trevni, since it does not have a knob to directly configure the row group size, we use $RG = \textit{max}$.

$f_{DP}$ is represented by *n-file*, which means the number of files (physical blocks) that a row group was packed into. Specifically, we used *1-file* and *16-file* in our experiments. For *16-file*, a column group only had a single column, i.e. for $f_{TP}$, $CG = \textit{non-grouped}$.

#### 4.2.3 *Storage Device Type*

A HDD was used because HDDs are commonly used in existing production deployments of HDFS. Also, we did not use RAID because in Hadoop clusters, Just a Bunch of Disks (JBOD) is the recommended configuration for slave nodes and using RAID is not recommended [9].

#### 4.2.4 *Accessing Patterns of the Application*

RCFile and Trevni both determine the size of the data needed in a read operation based on metadata. For reading a row group, RCFile loads data in a column by column way. For Trevni, we used both column by column and row by row methods. Additionally, to provide unbiased results, it is worth noting that when multiple columns were accessed, we chose those columns that were not stored contiguously. Also, in our experiments, the size of unneeded data between two needed columns was always larger than the size of a read buffer.

#### 4.2.5 *Processing Operations of the Application-level Filesystem*

We used two application-level filesystems provided by Hadoop, which were local filesystem (LFS) and distributed filesystem (DFS) [5]. For LFS, it uses buffered read operations and the buffer size is configurable. For DFS, it has two approaches to read data. First, if data is co-located with the client (the reader), the client can directly access the file through local OS. In HDFS, this approach is called

[4] 1 MiB $= 2^{20}$ bytes.

[5] LFS is a middleware on the top of the local OS. It provides the same set of interfaces as DFS

DFS short circuit. Second, clients will contact the server having the required data in HDFS to read data through socket. In this approach, buffered read operations are used.

On the aspect of the application-level asynchronous filesystem readahead, LFS does not have a mechanism to issue asynchronous requests to prefetch data which may be used in future. However, DFS can be configured to issue asynchronous requests to local OS through `posix_fadvise` [6]. Since this mechanism relies on local OS readahead, we disabled DFS readahead in our study and focused on the impact from local OS readahead.

#### 4.2.6 *Processing Operations of the Filesystem in OS*

For read operations from application-level filesystems or those directly from applications, local OS will execute these operations as requested. Also, local OS will issue asynchronous readahead requests in an adaptive way [26]. In our study, we used different maximum readahead sizes.

## 4.3 Benchmarking Tool

To have apple-to-apple comparisons, we have developed a benchmarking tool that uses RCFile and Trevni to simulate variations of each design factor in the scope of our study based on approaches discussed in Section 2.3. In this tool, all values of a table are serialized by the serialization and deserialization library in Hive. To focus on I/O read performance, this tool does not deserialize data loaded from underlying storage systems. To store a row of a column group, we use a composite data type (`STRUCT` in Hive) and this column group will be stored as a single column.

RCFile and Trevni have complementary merits. By combining them, our benchmarking tool covers a large variety of both table placement methods and behaviors of applications [7]. The main differences of these two implementations are summarized as follows. First, RCFile has a knob to explicitly configure the size of a row group, but Trevni does not have this flexibility. Second, when reading a column in a row group, RCFile reads the entire column at once. However, Trevni only reads a block (the unit for compression) [8] at a time. Third, when reading referenced columns in a row group, RCFile reads all needed columns in a column by column way before applications can access values of this row group, which means that RCFile determines the sequence of read operations issued from applications. However, Trevni does not have this restriction, and applications can determine how to read those referenced columns.

## 4.4 Eliminating Unnecessary Factors

We aim to only focus on the second, third, and fourth design factors summarized in Section 2.2.4. To accomplish this objective, we need to eliminate impacts from three unnecessary factors. First, columns with different data types can blur experimental results since different columns may have significantly different impacts on the I/O performance. To eliminate the impact of this factor, in our experiments, columns of a table had the same data type and all values had the same size (also discussed in Section 4.2.1). Second, optimizations existing in different implementations of table placement methods provide workload-dependent benefits and can make experimental results hard to reason. Because table placement methods studied in this paper are following the same basic structure presented in Section 2.1, they are equally affected by those optimizations. Thus, we did not use any optimization in our micro-benchmarks. Third, the variety of workloads can introduce unnecessary complexity and can cause biased results. In our experiments, if a column was referenced, it was read entirely because when only a portion of rows is needed, the initial ordering of rows directly affects the performance. However, in this way, our experiments cannot evaluate the performance of different table placement methods when indexes are used to access data in a physical block. We believe that the I/O issue of randomly accessing data in a physical block is beyond the scope of this paper and it will be studied in our future work.

It is worth explaining reasons why we did not consider compression. Because we focus on I/O performance, for a given table, compression techniques only shrink the size of this table by reducing (1) the size of a column in a row group and thus (2) the size of a row group. Thus, in our study, whether a table is compressed and how a column is compressed do not affect our results.

[6]http://linux.die.net/man/2/posix_fadvise

[7] Because we want to compare results of variations of each design factor in the scope of our study, it is not meaningful to compare results from RCFile-based simulations with results from Trevni-based simulations.

[8]The uncompressed size of a block is 64 KiB.

**Table 3: A summary of conventions of presenting different parameters in our experimental results.**

| Parameter | Explanation |
|---|---|
| `rg=x MiB` | The row group size was set to `x MiB` |
| `io=y KiB` | The size of read buffer was set to `y KiB` |
| `OS_RA=z KiB` | The maximum size of local OS readahead was set to `z KiB` |

## 4.5 Experimental Settings

In our micro-benchmarks, the machine we used had an Intel Core i7-3770K CPU and 32 GB memory. The OS was Red Hat Enterprise Linux Workstation 6.4 and the kernel release was 2.6.32-279.22.1.el6.x86_64. In our experiments, the HDD was a Western Digital RE4 1 TB SATA hard drive. We formatted the HDD with an ext4 filesystem and the HDD was mounted with options `-o noatime -o data=writeback`. The sequential bandwidth of this HDD measured through `hdparm -t` was around 130 MB/s. The version of Hadoop used in micro-benchmarks was 1.1.0. RCFile was distributed with Hive 0.9.0, and Trevni was distributed with Avro 1.7.3. Before every experiment, we first freed all cached objects, and then freed OS page cache, dentries and inodes.

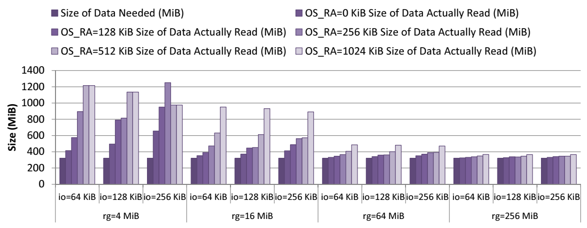
In our experiments, $T_1$ was stored in the local OS filesystem and it was accessed directly or through LFS. $T_2$ was stored in DFS and it was accessed through DFS with or without DFS short circuit. The reason that we stored a smaller table in DFS is that the size of the table is comparable to the size of the data processed by a Map task of Hadoop MapReduce. We configured DFS to use a large physical block size, so every file associated with $T_2$ was a single physical block (a HDFS block). If $T_2$ was stored in a single file, only a single physical block was used.

## 4.6 Experimental Results

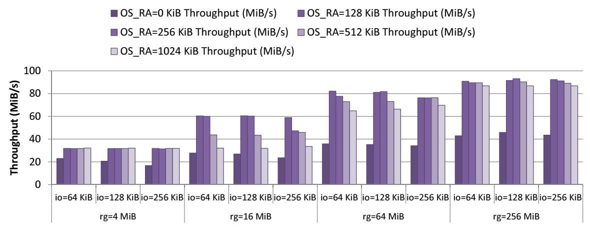
First, we summarize conventions of presenting different parameters in our experimental results in Table 3. We also follow these conventions in our macro-benchmarks (Section 5).

#### 4.6.1 *Row Group Size*

With a given table, the size of a row group determines the size of every column in this row group. Because data of a column stored in two row groups may not stored contiguously, if the size of this column in a row group is small, lots of seeks will be introduced and the performance of reading this column will be degraded.

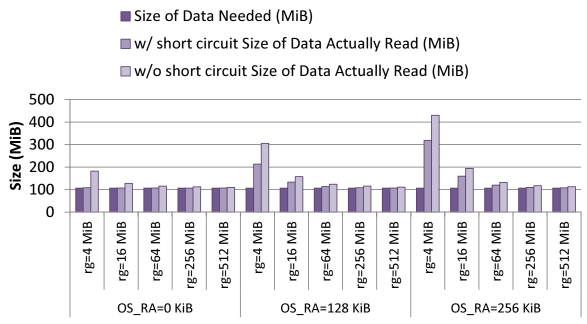


(a) Sizes when reading 4 columns

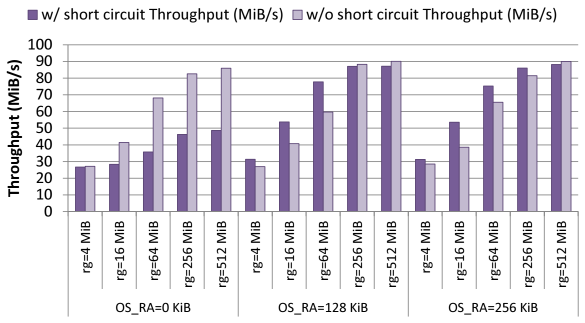


(b) Throughputs of reading 4 columns

**Figure 3: Sizes of needed data and actually read data, and throughputs on reading 4 columns of tables stored with (($*$, *non-grouped*), *1-file*, $T_1$) through LFS.**



(a) Sizes when reading 4 columns



(b) Throughputs of reading 4 columns

**Figure 4: Sizes of needed data and actually read data, and throughputs on reading 4 columns of tables stored with (($*$, *non-grouped*), *1-file*, $T_2$) through DFS. DFS has a 256 KiB read buffer.**

Figure 3 shows results of reading 4 columns of tables stored with (($*$, *non-grouped*), *1-file*, $T_1$) through LFS with different LFS read buffer sizes and different maximum local OS readahead sizes. We also did the same experiments on reading 1 and 2 columns, and we have the same observations as reading 4 columns.

Figure 3(a) shows sizes of data needed and sizes of data actually read from the HDD through LFS (collected by `iostat`). Through Figure 3(a), we have three observations.

First, we focus on results when local OS readahead is disabled (cases with the prefix of `OS_RA=0 KiB`). When a small row group

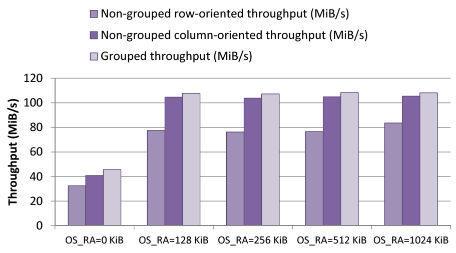


**Figure 5: Throughput of reading 2 columns from tables stored with ((*max*, *non-grouped*), *1-file*, $T_1$) and ((*max*, *grouped*), *1-file*, $T_1$) from the filesystem in the local OS.**

is used, a buffered read operation through LFS will read lots of unnecessary data from unneeded columns. For example, when the row group size was set to 4 MiB, in our benchmarks, the size of a column in a row group was implicitly set to 256 KiB. However, the size of a column in a row group cannot be exactly 256 KiB and it is actually slightly larger than 256 KiB. Thus, when the read buffer size was 256 KiB, the size of the data actually read from the storage device was around two times as large as the size of the data needed. Through Figure 3(a), we can see that the size of four columns was 320.4 MiB data. However, 655.5 MiB data was loaded from the storage device when settings mentioned above were used.

Second, if buffered read operations read both needed data and unneeded data from the device, local OS readahead will amplify the size of unneeded data since local OS will prefetch data from those unneeded columns. For example, when the row group size was 4 MiB and read buffer size was 256 KiB, to read four columns, more than half of the data in the table was loaded from the HDD when local OS readahead was enabled.

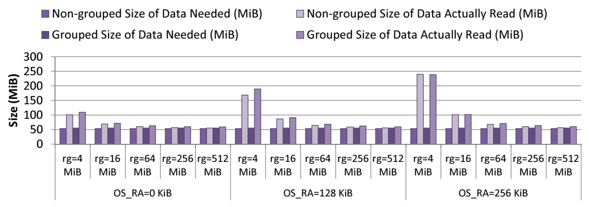
Third, with the increase of the row group size, there will be less buffered read operations that read unneeded data from the device, and local OS readahead will do less useless work. For example, when the row group size was 256 MiB, read buffer size was 256 KiB, and maximum local OS readahead size was 1024 KiB, to read four columns, 360.6 MiB data was actually read and the size of needed data was 320.4 MiB.

Figure 3(b) shows throughputs of reading data through LFS corresponding to Figure 3(a). When a read operation loads unnecessary data and OS readahead cannot efficiently prefetch needed data, the throughput of data reading is very low. For example, when the row group size was 4 MiB, read buffer size was 256 KiB, and maximum local OS readahead size was 1024 KiB, the throughput of reading four columns was 31.8 MiB/s. When the row group size was increased to 256 MiB, the throughput was 86.8 MiB/s.

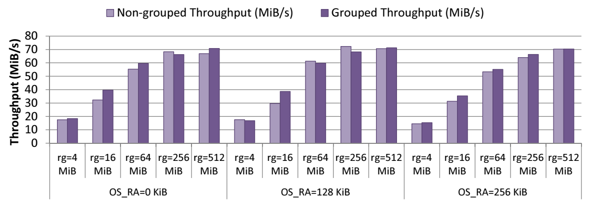
Figure 4 shows results when reading data of tables stored with (($*$, *non-grouped*), *1-file*, $T_2$) through DFS with 256 KiB read buffer size. We can see that sizes of data actually read and throughputs have the same trend as those presented in Figure 3. It is worth noting that, when DFS short circuit is enabled, RCFile does not use buffered read operations. Thus, when reading the metadata of a row group, it generates lots of small read operations (e.g. 1 byte request). This behavior is the reason that throughputs of cases with DFS short circuit were significantly lower than those cases without DFS short circuit when local OS readahead was disabled.

### 4.6.2 Grouping Columns

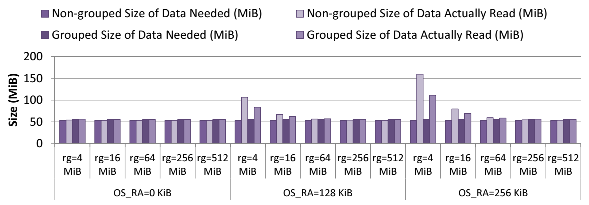
To present the impacts of grouping columns to column groups, we first compare results of reading two columns of tables stored with ((*max*, *non-grouped*), *1-file*, $T_1$) and ((*max*, *grouped*), *1-file*, $T_1$) directly from the filesystem in the local OS, which are shown in Figure 5. We configured Trevni to directly access local filesys-



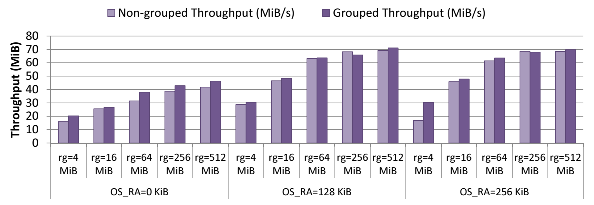
(a) Sizes without HDFS short circuit



(b) Throughputs without HDFS short circuit



(c) Sizes with HDFS short circuit



(d) Throughputs with HDFS short circuit

**Figure 6: Sizes of needed data and actually read data, and throughputs on reading 2 columns of tables stored with (($*$, *non-grouped*), *1-file*, $T_2$) and (($*$, *grouped*), *1-file*, $T_2$) through DFS. DFS has a 256 KiB read buffer.**

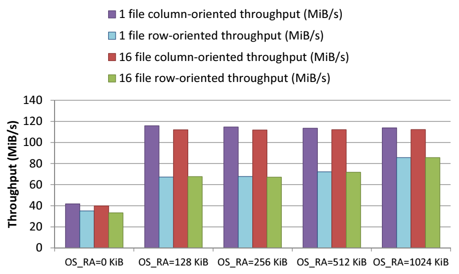
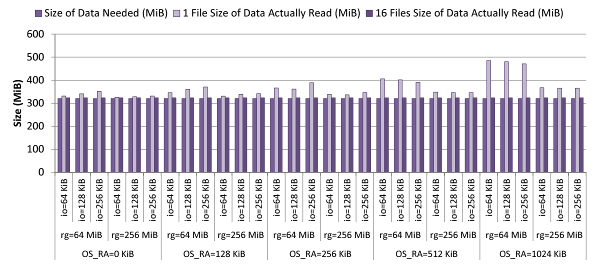
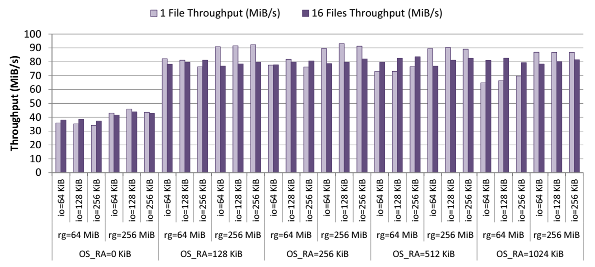


**Figure 7: Throughputs of reading 4 columns of tables stored with ((*max*, *non-grouped*), *1-file*, $T_1$) and ((*max*, *non-grouped*), *16-file*, $T_1$) from the filesystem in the local OS.**



(a) Sizes when reading 4 columns



(b) Throughput of reading 4 columns

**Figure 8: Sizes of needed data and actually read data, and throughputs on reading 4 columns of tables stored with (($*$, *non-grouped*), *1-file*, $T_1$) and (($*$, *non-grouped*), *16-file*, $T_1$) through LFS.**

tem instead of accessing files through LFS of DFS. Through this way, we can eliminate the impacts from buffered read operation introduced by LFS and DFS. When accessing the table that does not group columns, we used both the row-oriented access method and column-oriented access method. Through Figure 5, we can see that grouping columns has a significant advantage only when the row-oriented access method is used. Because row-oriented access method will generate lots of seeks among two columns to fetch the data, by grouping these two columns together, applications only read a single column group and thus, seeks are eliminated. However, compared with the column-oriented access method on non-grouped two columns, accessing two grouped columns does not have significant performance improvement (only 3 to 4 MiB/s improvement). This small difference on throughputs is caused by one additional disk seek and handling different metadata (in our experiments, the table stored with ((*max*, *non-grouped*), *1-file*, $T_1$) had 16 columns and the table stored with ((*max*, *grouped*), *1-file*, $T_1$) had 8 columns).

Additionally, we used tables stored with (($*$, *non-grouped*), *1-file*, $T_2$) and (($*$, *grouped*), *1-file*, $T_2$) to study combined impacts of different row group sizes and grouping columns. Figure 6 shows results on DFS when accessing two columns of a table with and without HDFS short circuit, varied row group sizes, and different maximum local OS readahead sizes. From cases with and without DFS short circuit, we can see that, when a small row group size is used, grouping columns can have performance improvement. Without DFS short circuit, buffered read operations are used. In this case, grouping columns can increase the size of needed data in a read buffer when a small row group size is used. Also, when a small row group is used, local OS readahead can easily do useless work by prefetching unneeded columns. By grouping columns, local OS readahead can do more useful work. However, when a large row group size is used, grouping columns does not yield significant performance improvement. The limited improvements over non-grouped cases are caused by saving a limited number of disk seeks and handling different metadata.

### 4.6.3 Packing Column Groups

Storing a row group to multiple physical blocks introduces a new problem on how to group columns or column groups into physical blocks. For simplicity, when columns in a row group were packed into multiple physical blocks, we stored a single column in a physical block.

To show impacts of packing columns in a row group into multiple physical blocks, we first present results of accessing 4 columns of tables stored with ((*max*, *non-grouped*), *1-file*, $T_1$) and ((*max*, *non-grouped*), *16-file*, $T_1$) directly from the filesystem in the local OS in Figure 7. We can see that, for both the row-oriented access method and column-oriented access method, using multiple physical blocks to store a row group does not provide significant performance improvement over corresponding cases that store all columns in a physical block.

We also present results of reading 4 columns from tables stored with (($*$, *non-grouped*), *1-file*, $T_1$) and (($*$, *non-grouped*), *16-file*, $T_1$) through LFS in Figure 8. Through results, we can see that packing columns into different physical blocks only have performance advantages when a smaller row group size is used and a relative large maximum local OS readahead size is used. In this configuration, when a single physical block (a single file in the local OS) is used to store a row group, the local OS can do lots of useless work by prefetching data from unneeded columns because it is not aware of boundaries between columns. When a large row group size is used, storing columns in different physical blocks does not have performance advantages over storing all columns in one file. We also evaluated performance of reading 4 columns from tables stored with (($*$, *non-grouped*), *1-file*, $T_2$) and (($*$, *non-grouped*), *16-file*, $T_2$) through DFS. Results of these experiments show the same trend as Figure 8.

## 4.7 Summary

With our thorough study and insightful findings, we suggest three general-purpose and effective actions on how to adjust table placement methods to achieve optimized I/O read performance.

### 4.7.1 Row Group Size

**Action 1: using a sufficiently large row group size**. As shown in our experimental results, the size of a row group should be sufficiently large. Otherwise, a buffered read operation issued by the underlying storage system can load more data from unneeded columns and the disk seeking cost cannot be well amortized.

In a distributed environment, determining the suitable row group size requires careful considerations on the single machine data reading performance and the available degree of parallelism. We will discuss these considerations in Section 6.

### 4.7.2 Grouping Columns

**Action 2: if a sufficiently large row group size is used and columns in a row group can be accessed in a column-oriented way, it is not necessary to group multiple columns to a column group**. Once two or more columns are grouped, reading a subset of columns stored in this column group requires to load the data of the entire column group from the underlying system, and thus, unneeded data will be read. Also, because of the variety of workloads, determining how to group columns may also introduce burden and overhead on grouping columns.

If applications have to access columns in a row group in a row-oriented way, it may worth grouping columns to column groups. For example, if all columns in a column group are needed, this column group can be read sequentially.

### 4.7.3 Packing Column Groups

**Action 3: if a sufficiently large row group size is used, it is not necessary to pack column groups (or columns) into multiple physical blocks**. When a sufficiently large row group size is used, the negative impacts from local OS asynchronous readahead are negligible. Also, if columns in a row group need to be placed in the same machine, packing column groups into multiple physical blocks may require additional supports from the underlying system, which may not be feasible and can introduce overhead.

**Table 4: A summary of datasets we used in our macro-benchmarks. The column of Scale Factor shows the scale setting we used to generate every dataset, and the column of Total Size shows the corresponding size of the entire dataset.**

| Benchmark | Scale Factor | Total Size |
|---|---|---|
| SSB | 1000 | 1000 GB |
| TPC-H | 1000 | 1000 GB |
| SS-DB | *large* | 9.9 TB |

If the maximum size of a row group is limited by the data processing environment and the negative impacts from local OS asynchronous readahead are observable, it may worth packing column groups into multiple physical blocks, which can indirectly enlarge the size of a row group. In this case, the issue of co-locating physical blocks that store column groups in a row group is still needed to be carefully considered and addressed.

## 5. MACRO-BENCHMARK RESULTS

In this section, we present results of a set of macro-benchmarks to show impacts of different row group sizes (Section 5.3.1) and impacts of grouping columns (Section 5.3.2) in production workloads. We do not present results of packing column groups of a row group into multiple physical blocks at here because of two reasons. First, packing column groups of a row group into multiple physical blocks prevents local filesystem from prefetching unnecessary data. The impact of it has been studied clearly in Section 4.6.3. Second, in existing systems, packing column groups in a row group into more than one physical blocks is not practical because it increases the memory usage of the master node (e.g. NameNode in HDFS), and it requires customized block placement policy to achieve expected performance.

## 5.1 Workloads

In our macro-benchmarks, we evaluated queries selected from Star Schema Benchmark (SSB) [18], TPC-H [10], and SS-DB (A standard science DBMS benchmark) [17]. In our evaluation, we generated a large dataset for every benchmark. Table 4 shows the scale factor we used and the size of every dataset. Queries used in our experiments are summarized as follows.

**SSB Query 1.1, 1.2, and 1.3:** Every query in these three queries first has a join operation between a fact table (`lineorder`) and a dimension table (`date`), and then has an aggregation operation on results of the join operation. These queries have different filter factors (the ratio of needed rows to the total number of rows) on the fact table. In our results, query 1.1, 1.2, and 1.3 are referred to as ssb.1.1, ssb.1.2, and ssb.1.3, respectively.

**TPC-H Query 6:** This query is an aggregation query on the table of `lineitem`, which is the largest table in TPC-H.

**SS-DB Query 1.easy, 1.medium, and 1.hard:** These queries are aggregation queries on images. These three queries differ from each other on the difficulty, i.e. easy, medium, and hard. With the increase of the difficulty (e.g. from easy to medium), the volume of data that will be processed in the query increases. It is worth noting that queries in SS-DB processed only one cycle of images. In our dataset, one cycle of images has a raw size of around 200 GB. In our results, query 1.easy, 1.medium, and 1.hard are referred to as ssdb.q1.easy, ssdb.q1.medium, and ssdb.q1.hard, respectively.

These queries were selected based on two reasons. First, they are from widely used benchmarks. Second, they can be executed in a single MapReduce job and thus make it easier to reason impacts of different table placement methods on real applications than complex queries. If we use queries that need to be evaluated by multiple

MapReduce jobs, we would introduce more operations which are invariant to different choices of table placement methods. Those irrelevant operations will equally contribute to elapsed times in our experiments. Thus, it is not reasonable to use queries that need to be evaluated by multiple MapReduce jobs in this paper.

## 5.2 Experimental Settings

We used Hive 0.11.0 running on Hadoop 1.2.0 in our macro-benchmarks. Because Trevni has not been integrated into Hive, we only used the RCFile-based simulation. We used Hive in a 101-node Hadoop cluster, which was setup in Amazon EC2. This cluster had one master node and 100 slave nodes. The virtual machine (VM) instance of the master node was `m1.large`. The VM instance of 100 slave nodes was `m1.medium`. In this cluster, the master node ran the NameNode and the JobTracker. Each slave node ran a DataNode and a TaskTracker.

HDFS, MapReduce and the local filesystem in every node were setup as follows. For HDFS, we set replication factor to 3 and HDFS block size to 256 MiB. For MapReduce, we set every slave node to have 1 Map slot and 1 Reduce slot based on specifications of the slave node. We also set that a spawned JVM can be reused unlimited times. To measure exact elapsed times of a Map phase and a Reduce phase, the Reduce phase was started after all Map tasks had finished. For the local filesystem, we set the maximum local OS readahead sizes to 512 KiB, which is a recommended size for Hadoop clusters [11].

For every experiment, we report the median of three repetitions. Before every repetition of our experiments, we first freed all cached objects, then freed OS page cache, dentries and inodes.

## 5.3 Experimental Results

Our experimental results of macro-benchmarks are presented in Figure 9. In our experiments, we used 4 MiB, 64 MiB, and 128 MiB as row group sizes. We used 64 KiB and 128 KiB as read buffer sizes. We used two schemes to group columns. In figures presented in this section, `GroupingColumn=no` means that we did not group columns and we stored a single column in a column group. `GroupingColumn=Best` means that we stored all needed columns of a query into a single column group.

### 5.3.1 *Row Group Size*

Based on Figure 9, we can see that when the row group size is small, Hive will read significantly more data than needed for all queries from three benchmarks. With the increase of row group size, the size of the data actually read decreases significantly.

Additionally, elapsed times of Map phases are affected by different row group sizes. From 4 MiB row group size to 64 MiB row group size, the elapsed times of the Map phase decrease significantly. However, when further increase the row group size from 64 MiB to 128 MiB, the change on the elapsed times of the Map phase is not significant. It is because 64 MiB row group size is large enough for these workloads. Any further increase of the row group size only result in a small gain of query processing performance.

### 5.3.2 *Grouping Columns*

When the row group size is small, grouping columns can provide benefits in two aspects. First, because needed columns are stored together, a buffer read operation can load a less amount of unnecessary data from disks. Second, a less number of disk seeks is needed to read needed columns. However, when the row group size is large enough, grouping columns cannot provide significant performance benefits. Based on Figure 9, for 64 MiB or 128 MiB row group size, there is no significant difference between storing a single column in a column group and the best column grouping scheme which stored only needed columns in a column group.

## 5.4 Summary

From experimental results of our macro-benchmarks, we can confirm that our observations of our micro-benchmarks also exist in a production-like environment. Moreover, our suggested actions based on results of micro-benchmarks are also valid in practice.

## 6. THE DATA READING EFFICIENCY AND THE DEGREE OF PARALLELISM

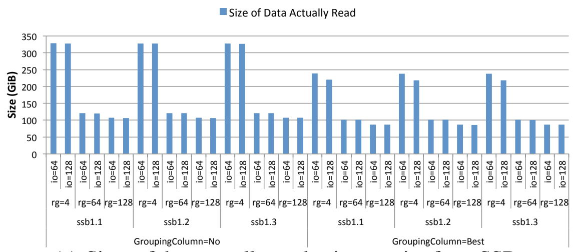
As shown in our experimental results, a sufficiently large row group size can make read operations efficient on HDDs. In recent work, e.g. ORC File [4] and Parquet [5], a large row group size is also recommended. However, in a distributed environment, increasing the row group size may not always result in performance improvement. On one hand, for a given table, using a large row group size can improve the efficiency of data reading because more values of a column (or a column group) in a row group can be stored contiguously, which improves the efficiency of disk read operations. Cost-efficient read operations are highly desirable, e.g. in a cloud environment like Amazon Elastic MapReduce [12], a higher data reading efficiency implies a lower price-performance ratio. On the other hand, a smaller row group size can increase the available degree of parallelism. A high degree of parallelism has three benefits. First, more tasks can be used to process data in parallel, which can increase the aggregated data reading throughput. Second, tasks assigned to machines can achieve better dynamic load balance, which reduces elapsed time cased by unbalanced loads. Third, when a machine fails, more available ones can be used to process tasks originally assigned to the failed one, which decreases the recovery time in the presence of machine failure events.

The essential issue to be considered is the trade-off between the degree of parallelism in a cluster and the data reading efficiency in each individual machine (or each data processing task). Users having different tables, infrastructures and requirements on costs should choose different row group sizes with considerations of the above trade-off. Here we provide an example to illustrate our point of view. The table in this example is based on `lineitem` in TPC-H benchmark [10] and we call this table $L$. Values from the same column in $L$ have the same size. The size of a value in a column is the average value size of the corresponding column in `lineitem` calculated based on TPC-H specification (revision 2.15.0) and the actual data type used in Hive [13]. This table $L$ has four integer columns, four double columns, two 1-byte string columns, four 10-byte string columns, one 25-byte string column, and one 27-byte string column. We assume that a value in an integer column occupies four bytes and that a value in a double column occupies eight bytes.

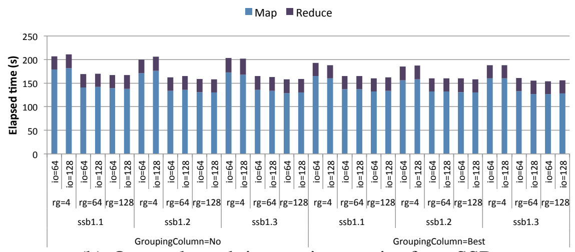
In this example, we assume that a logical subset in a row group only has values of a single column. We then analyze the change of data reading efficiency on reading a single column of a row group with the increase of the row group size. The data reading efficiency is calculated as the ratio of the data reading throughput to the disk bandwidth. Equation 4 shows how we calculate the data reading efficiency with a given size of a column in a row group ($S_C$), disk seek time ($t_{\text{seek}}$), and disk bandwidth ($B_D$).

$$\text{Efficiency} = \frac{\text{Throughput}}{B_D} = \frac{S_C}{t_{\text{seek}} + \frac{S_C}{B_D}} \times \frac{1}{B_D}, \qquad (4)$$

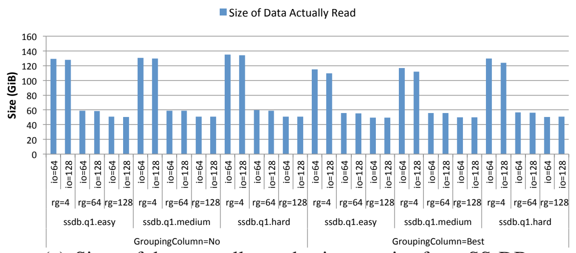
Figure 10 shows the data reading efficiency on reading different columns with the increase of the row group size. In this analysis,



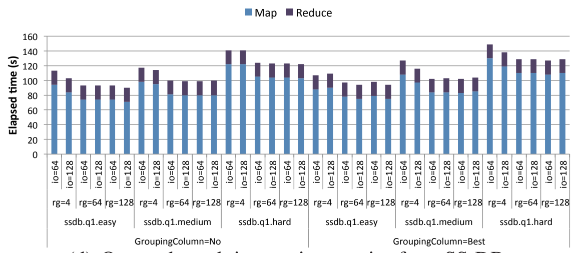
(a) Sizes of data actually read using queries from SSB



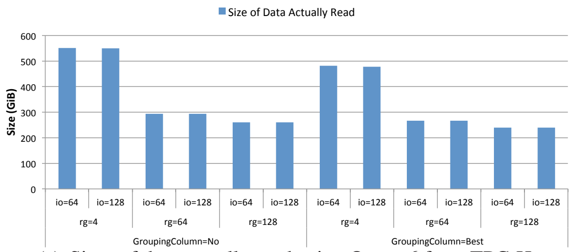
(b) Query elapsed times using queries from SSB



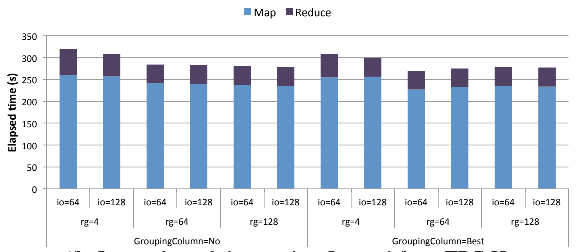
(c) Sizes of data actually read using queries from SS-DB



(d) Query elapsed times using queries from SS-DB



(e) Sizes of data actually read using Query 6 from TPC-H



(f) Query elapsed times using Query 6 from TPC-H

**Figure 9: Sizes of data actually read and elapsed times of queries introduced in Section 5.1. `GroupingColumn=x` represents the scheme of grouping columns. The unit of the row group size (`rg`) is MiB. The unit of the read buffer size (`io`) is KiB.**

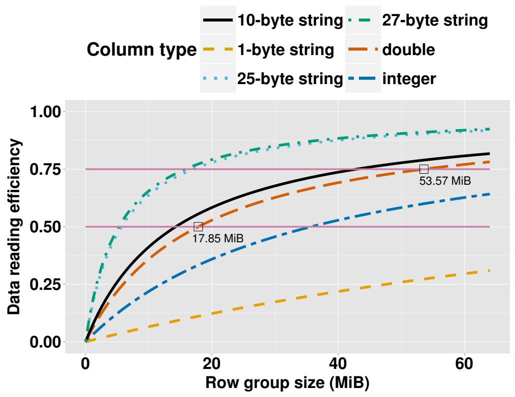


**Figure 10: The change of data reading efficiency with the increase of the row group size.**

we set $t_{\text{seek}} = 10$ ms and $B_D = 100$ MiB/s. For the purpose of simplicity, we do not consider the impact from buffered read operations. Because different columns have different sizes of values, they have different data reading efficiency for a given row group size.

We summarize the implications presented in Figure 10 as follows. First, when we start to increase the row group size from a small number, the increase of data reading efficiency of those columns with small value sizes increases much slower than those columns with large value sizes. For the smallest columns (those two 1-byte string columns), when the row group size is 64 MiB, the efficiency is only 31%. However, for the 27-byte string column, it can achieve 75% efficiency when the row group size is less than 20 MiB. Second, the increase of data reading efficiency via enlarging the row group size decreases when the row group becomes bigger. Taking columns with type double as an example, to achieve 50% and 75% data reading efficiency, 17.85 MiB and 53.57 MiB row group sizes should be used, respectively (hollow squares in the figure). Third, for a table with columns having a high variance of average value sizes, achieving a certain data reading efficiency goal for all columns may not be practical because a very large row group size can limit the degree of parallelism. We conclude that an optimal row group size is determined by a balanced trade-off between the data reading efficiency and the degree of parallelism.

## 7. ORC FILE: A CASE STUDY

Recently, Optimized Record Columnar File (ORC File) has been designed and implemented to improve RCFile. In this section, we explain its design from the perspective of our study as a case study. Specifically, we describe ORC File from the aspect of the basic structure of table placement methods, i.e. row reordering, table partitioning, and data packing. In the end, we also highlight other improvements in ORC File.

**Row reordering:** The writer of ORC File does not sort a table, but users can explicitly sort it as a pre-process. Like other table placement methods, a sorted table may improve the efficiency of data encoding and compression. Also, ORC File supports predicate pushdown [8] [14]. The improvement of performance contributed by this feature on reading a sorted column is more significant because the reader of ORC File can effectively skip rows which do not satisfy the given predicates.

**Table partitioning:** On table partitioning, ORC File is similar to RCFile. For primitive data types, the function of table partitioning of RCFile is also applicable to ORC File. However, ORC File uses a 256 MiB row group size by default (a row group is called by a stripe in ORC File). This large stripe size improves the performance of reading data, which is consistent to our suggested action in Section 4.7.1. Since the default stripe size is large, as we discussed in Section 6, it is possible that the degree of parallelism on processing a given table is bounded by the stripe size [15]. ORC File takes this issue into consideration in its optimization. Also, ORC File does not group columns, which is consistent with our suggested action in Section 4.7.2. For complex data types like `Map`, ORC File decomposes a column with a such data type to multiple actual columns with primitive data types [16]. In contrast, RCFile does not decompose a column with a complex data type.

**Data packing:** ORC File does not store a stripe (a row group) to multiple files. Because the size of a stripe is usually large and an ORC File reader uses the column-oriented access method, it is not necessary to store a stripe to multiple files from the perspective of I/O read performance. Also, storing all columns of a stripe in a single file is friendly to the NameNode of HDFS. This design choice is consistent with our suggested action in Section 4.7.3.

ORC File also have several major improvements on auxiliary data and optimization techniques. On the aspect of auxiliary data, ORC File has integrated indexes that support rapidly seeking to a specific row (identified by the row number) and predicate pushdown. In contrast to RCFile, ORC File records the locations of stripes in the footer. Thus, the reader of ORC File does not need to scan the file to locate the starting point of a stripe. On the aspect of optimization techniques, ORC File uses two levels of compression. The writer first automatically applies type-specific encoding methods to columns with different data types. Then, an optional codec can be used to compress encoded data streams. Interested readers may refer to [8] for details of these improvements.

## 8. CONCLUSION

We have presented our study on the basic structure and essential issues of table placement methods in Hadoop-based data processing systems. We have proposed a general abstraction framework that makes it possible to abstract and compare different table placement methods in a unified way. Based on this framework, we have developed a benchmarking tool that simulates variations of table placement methods. We have conducted a set of comprehensive experiments with our benchmarking tool to understand the fundamental impacts of different variations of each design factor. Experimental results from our large-scale experiments have also confirmed results of micro-benchmarks in production workloads.

Our main findings are four-fold: (1) the row group size should be large enough so that the column (or column group) size inside a row group will be large enough to facilitate applications to achieve efficient read operations; (2) 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; (3) when the row group size is large enough, it is not necessary to store columns or column groups in a row group to multiple physical blocks; (4) in a distributed computing cluster, the row group size should be selected with considerations on the trade-off between the data reading efficiency in each machine or task and the degree of parallelism. Our presentation of ORC File makes a case on the effectiveness of our findings and suggested actions.

Our benchmarking tool used in micro-benchmarks and scripts used in macro-benchmarks are available at https://github.com/yhuai/tableplacement.

## 9. ACKNOWLEDGMENTS

We thank anonymous reviewers for their constructive comments. This work was supported in part by the National Science Foundation under grants of CNS-1162165 and OCI-1147522.

## 10. REFERENCES

[1] https://hive.apache.org/.
[2] https://pig.apache.org/.
[3] https://github.com/cloudera/impala.
[4] https://issues.apache.org/jira/browse/HIVE-3874.
[5] https://github.com/Parquet/parquet-format.
[6] http://avro.apache.org/docs/current/trevni/spec.html.
[7] http://avro.apache.org/.
[8] http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.0.0.2/ds_Hive/orcfile.html.
[9] http://docs.hortonworks.com/HDPDocuments/HDP1/HDP-1.3.0/bk_cluster-planning-guide/content/hardware-for-slave.html.
[10] http://www.tpc.org/tpch/.
[11] http://www.slideshare.net/hortonworks/virtualize-hadoop050113webinar.
[12] http://aws.amazon.com/elasticmapreduce/.
[13] https://issues.apache.org/jira/browse/HIVE-600.
[14] https://issues.apache.org/jira/browse/HIVE-4246.
[15] https://issues.apache.org/jira/browse/HIVE-4868.
[16] http://www.slideshare.net/oom65/orc-andvectorizationhadoopsummit.
[17] SS-DB: A Standard Science DBMS Benchmark. http://www-conf.slac.stanford.edu/xldb10/docs/ssdb_benchmark.pdf.
[18] Star Schema Benchmark. http://www.cs.umb.edu/~poneil/StarSchemaB.PDF.
[19] S. Chen. Cheetah: A High Performance, Custom Data Warehouse on Top of MapReduce. In *VLDB*, 2010.
[20] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. Column-Oriented Storage Techniques for MapReduce. In *VLDB*, 2011.
[21] S. Guo, J. Xiong, W. Wang, and R. Lee. Mastiff: A Mapreduce-based System for Time-Based Big Data Analytics. In *CLUSTER*, 2012.
[22] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems. In *ICDE*, 2011.
[23] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. Trojan Data Layouts: Right Shoes for a Running Elephant. In *SOCC*, 2011.
[24] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. YSmart: Yet Another SQL-to-MapReduce Translator. In *ICDCS*, 2011.
[25] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi, and S. Wu. Llama: Leveraging Columnar Storage for Scalable Join Processing in the Mapreduce Framework. In *SIGMOD*, 2011.
[26] F. Wu, H. Xi, and C. Xu. On the Design of a New Linux Readahead Framework. *SIGOPS Oper. Syst. Rev.*, 42(5):75–84, July 2008.
[27] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and Rich Analytics at Scale. In *SIGMOD*, 2013.