A Lightweight Evaluation Framework for Table Layouts in MapReduce Based Query Systems

Feng Zhu[1,2], Jie Liu[2,3], Lijie Xu[1,2], Dan Ye[2], Jun Wei[2,3], and Tao Huang[2]

[1] University of Chinese Academy of Sciences
[2] State Key Laboratory of Computer Sciences
[3] Institute of Software, Chinese Academy of Sciences
{zhufeng10, ljie, xulijie09, yedan, wj, tao}@otcaix.iscas.ac.cn

Abstract. Table layout determines the way how the relational row-column data values are organized and stored. In recent years, considerable candidates have been developed in MapReduce based query systems; they differ on storage space utilization, data loading time, query performance and so on. In most time, users are confronted with the problem of choosing the comprehensive optimum table layout given the workloads and the schema of tables. The straightforward way to run queries on generated data and compare the results is time consuming, and incurs the inaccuracy due to the MapReduce's nondeterministic execution runtime. In this paper, we propose a lightweight framework to evaluate table layouts without running the query. The framework adopts the *black box method* to test critical metrics, and the *query aware strategy* that extracts table-layout-related operations from query. Based on the metrics and operations, the framework makes suggestions to users. We conduct extensive experiments to empirically study the popular table layouts. Through the results illustration, we discover that column projection and compression are the most two prominent factors for general cases. Moreover, we discuss optimization chances for the intermediate tables produced in high level language systems.

Keywords: Table Layout, MapReduce, Query Aware, Black Box, Performance

1 Introduction

The MapReduce [6] programming model and its open-source implementation Hadoop [1] has now become the *de facto* framework to store and process massive data. However, it is usually too complicated to express complex analytical tasks (e.g., the business intelligence) as primitive MapReduce jobs. To enhance productivity, high level systems, that brings the relational concepts like SQL, table, row and column, have been built, such as Pig [8] and Hive [7].

In such environment, data is managed as the structured relational table that consists of rows and columns. Table layout determines the way how the two dimensional data values will be organized in the underlying distributed file system. For example, the one based on Hadoop's built-in *textfile* encapsulates table rows into records; properties, like the field delimiter, need to be explicitly specified to store and resolve tables. Besides, table layouts with sophisticated mechanisms have also been proposed, such as Zebra [5], CFile in Llama [10], CIF [11], RCFile [9], ORCFile [2], Parquet [3], Trevni [4], Trojan [13] and SLC-Store [12].

However, the different table layouts are proposed and implemented independently; a comprehensive study to compare them has not been done [14]. In practice, many situations involve evaluation on table layouts for appropriate choice, for example the following two scenarios. (1) *Table Configuration in Design Phase.* It is common for users to configure tables' layouts under specific workloads in the design phase. Migrating tables and queries from relational database to Hadoop is a representative case. In such case, the schema of tables and the queries are known, appropriate table layouts can achieve comprehensive advantages on data loading, query performance and storage space. (2) *Adaptively Set Intermediate Table Layout.* High level language systems translate queries into MapReduce job workflows. The temporary intermediate tables produced by the previous job will be consumed as input by subsequent job in the workflow. For the performance improvement, can we adaptively set the intermediate tables' layouts rather than the default setting (for example, the *sequencefile* in Hive)? Above all, the problem can be abstracted as: *Given workloads and the schema of tables, make suggestion on appropriate table layout.*

The straightforward solution is to run queries on generated or sample data, and then compare the results. However, it has many defects. To begin with, the query's MapReduce runtime in distributed environment is nondeterministic. Even the time costs on performing the same query with the same configuration twice are always different. This nondeterministic behavior is caused by many table-layout-independent factors (like scheduling), and incurs the inaccuracy of evaluating the real performance of table layouts. Moreover, it is too time-consuming to run all queries; actually, table layouts are aware of only few operations even in a complex query. Another way for the problem is to establish fine-grained performance models that cover all factors for each table layout. The challenges, which will be illustrated later, result that common performance models are unrealistic to establish.

To address the problem, we propose and implement an evaluation framework. (1) The framework, from the perspective of a task (stack: "$query \rightarrow workflow \rightarrow job \rightarrow task$") that directly interacts with table layouts, is lightweight. (2) The framework evaluates high-level user-oriented metrics, rather than table layouts's internal implementation factors, and provides users with comprehensive evaluation report. (3) The framework adopts the testing approach which is based on the *black box method* and the *query aware strategy.* Different table layouts are treated as black boxes with configuration knobs. The query aware strategy extracts the table-layout-related operations from upper computing layer's query.

Contributions. We identify three high-level user-oriented metrics of table layouts. Based on the black box method and query aware strategy, we design and implement a practical lightweight framework to evaluate different table layouts. From the perspective of a task, the framework avoids the nondeterministic behavior and the time cost of running the query. It automatically extracts table-layout-related operations from queries and rapidly evaluates the metrics. Moreover, we conduct extensive experimental studies to make empirical analysis on representative table layouts and discuss the optimizations chances for intermediate tables.

The reminder of this paper is organized as follows. In section 2, we introduce the related works. Section 3 presents the black box method and the query aware strategy. In section 4, we describe the evaluation framework. Section 5 conducts experimental study and makes discussion. We conclude our work in section 6.

2 Related Work

As early as the development era of relational database systems, table layouts had been widely studied. For example the row-oriented store [15], the column-oriented store [16,17,18] and the hybrid PAX store [19]. The weaknesses and advantages for each table layout were intensively investigated. In [18], D.J. Abadi et al makes an in-depth discussion among them.

In Hadoop's distributed environment, table layout has also attracted a wide range of interests in both academia and industry. However, these table layouts are proposed and implemented independently or even for specific workloads. Most recently, Yin Huai et al. [14] makes a comprehensive and systematic experimental study; they define three core operations to abstract table layouts' behaviors and evaluate key factors: (1) table's horizontal logical subset size, (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. Based on the evaluation, practical actions to optimize read performance are suggested. Our work differs from it as follows. To begin with, we argue that the abstraction of three core operations to describe the table layouts has its limitations. It is difficult to unify all table layouts and the three factors are not common. For example, CIF has no concept of row group and column group. Our work covers both the read and write performance. Moreover, we consider computing layer's queries and implement a lightweight evaluation framework for practical use.

Related works for query optimization are usually conducted from two layers: the translator and the MapReduce computing primitive. In the MapReduce primitive layer, the study focuses on MapReduce's performance models and some common optimization techniques. For example, H. Herodotou [20] proposes cost models to analyze and optimize MapReduce programs. MRShare [21] investigates opportunities to reduce the number of MapReduce jobs, i.e., the so-called "vertical packing"; while Stubby [22] further covers "horizontal packing" opportunities between two MapReduce job workflows. Particularly, optimizing the SQL-to-MapReduce translator also gains broad attentions and several rule-based translators has emerged in recent years, such as YSmart [23]. It applies a set of rules to use the minimal number of MapReduce jobs to execute multiple correlated operations in a complex query.

3 Black Box Method and Query Aware Strategy

3.1 Table Layout Insight

We take four representative table layouts as examples: TextFile, SequenceFile, RCFile and ORCFile, which include both unstructured and structured layouts and cover various performance-related factors.

- **TextFile.** The ASCII encoded text based file format in HDFS. The MapReduce job reads one line at a time and returns the byte offset as the *key* and the line of text as the *value*, with encapsulating a table row.
- **SequenceFile.** SequenceFile provides a persistent append-only data structure for binary key-value pairs. For table layout, the key is null and the value encapsulates a table row.
- **RCFile.** RCFile stores columns of a table in a record columnar way. It first partitions rows horizontally into row splits, and then vertically partitions each row split in a columnar way.
- **ORCFile.** The optimized version of RCFile. Compared with RCFile, it provides mechanisms such as fine-grained column encoding schemes, predicate push down an so on.

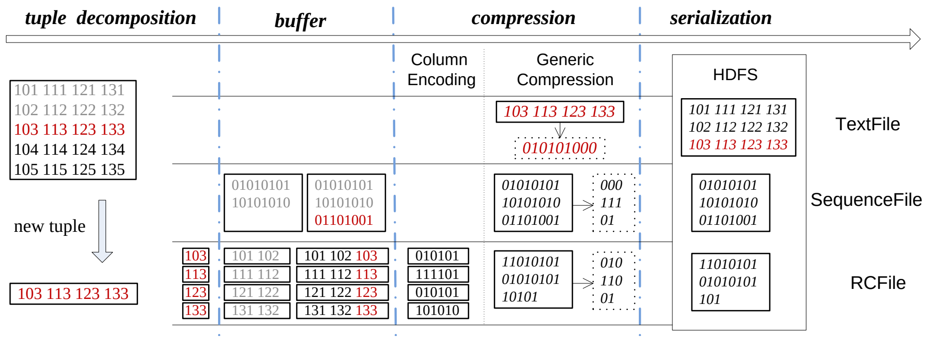


Fig. 1. Common Procedures for Writing

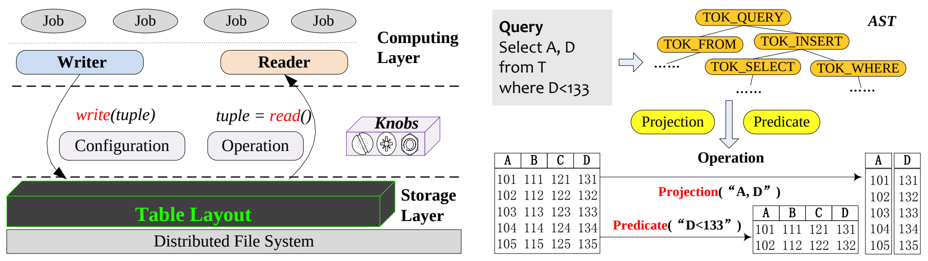
Fig.1 demonstrates four optional consecutive phases (i.e., decomposition, buffer phase, compression and serialization) for writing a tuple into the file system (the case of ORCFile is similar to that of RCFile). Correspondingly, the common phases for reading records are reversed, i.e., deserialization, decompression, buffer phase and reconstruction. For simplified presentation, we only illustrate the writing behaviors here.

Tuple decomposition is the first phase for those none row-oriented table layout. In this phase, the tuple will be decomposed as different concepts like columns, column groups and so on. Before flushed to disk, data will be first managed in the buffer. The buffer not only stores data, but also maintains the metadata for data. The metadata stores the information such as column data size, column type and even statistics (like max, min and etc.) in sophisticated table layouts. When the data reaches the configured capacity, it will be written out to the disk. The compression phase consists of two levels, the column encoding schemes and the generic compression algorithms. The column encoding scheme adopts fine-grained algorithms for different column types, for example, the run length encoded algorithm for integers and the dictionary for strings. The generic compression algorithms take table as common binary files using algorithms like gzip, zlib and so on. Compression results in lower IO cost at the expense of higher CPU computing. Serialization converts in-memory data structures into bytes that can be transmitted over the network or flushed to the disk. The serialization phase is common with all table layouts.

The above insights reveal difficulties to establish fine-grained performance models for table layouts. (1) It is too complex for the models that covers all factors and table layout's configuration parameters, resulting that models are hard to be accurate. (2) Fine-grained models are difficult to validate and more importantly, the internal factors are meaningless to users and the computing layer. (3) The performance of table layout is closely related with implementations. In such case, comparisons among them are apple to an orange.

3.2 Black Box Method

Throughout this paper, we adopt the testing approach based on the black box method to evaluate the table layouts. As depicted in the left part of Fig.2, the computing layer of MapReduce based systems write and read data row by row, respectively through the unified *write()* and *read()* interfaces.



A	B	C	D
101	111	121	131
102	112	122	132
103	113	123	133
104	114	124	134
105	115	125	135

A	D
101	131
102	132
103	133
104	134
105	135

A	B	C	D
101	111	121	131
102	112	122	132

Fig. 2. Black Box Method and Query Aware Strategy

The black box method treats different table layouts as different boxes with knobs that can be adjusted by users to define configuration parameters. The knobs for writing are key/value pairs (i.e., buffer size, row group size and etc) stored in the *conf*. Based on these specified settings, the layout determines how the record data will be organized. The knobs for records reading are projection and predicate, with which the IO cost can be reduced by pushing down the operations into the storage layer. However, the two knobs are optionally implemented and not exposed by all table layouts; for example, the RCFile only supports projection.

3.3 Query Aware Strategy

It is unnecessary to evaluate a table layout through executing the query. Actually, table layouts are aware of only few operations even in a complex query; the most query execution time is spent on in-memory computations. Rather than executing the query, the query aware strategy only extracts the table-layout-related operations.

Table-layout-related operations extraction. Given a specific query, the strategy translates it into abstract syntax tree (AST) and then analyzes AST to extract the operations that can be pushed down to table layouts. In MapReduce based query systems, only the operations of *column projection* and *predicate push down* can be exploited by the computing layer. The process of extracting these two operations from AST is straightforward, the algorithm searches the tree in a depth-first behaviour and gets the operations in the nodes with the corresponding keywords (e.g, the TOK_SELECT, TOK_WHERE and so on).

As demonstrated in the right part of Fig.2, the column projection ("A,D") and the predicate ("$D < 133$") are extracted from the query. Assume query Q has m times scans on table T, we denote the operations as the set of projection and predicate tuples, $OP(Q,T) = \{< projection_i, predicate_i > |1 \leq i \leq m\}$; for the example query, the result is $OP(Query, T) = \{< (A, D), (D < 133) >\}$.

4 TLEF: Table Layout Evaluation Framework

4.1 Design Methodology and Evaluation Metrics

The task (i.e., *mapper* and *reducer*), which directly interacts with table layout, is the bottom unit of the stack "$query \rightarrow workflow \rightarrow job \rightarrow task$". We advocate "see big things through small ones", and propose TLEF to evaluate the critical metrics of table layout from the perspective of a task.

Rather than table layouts' detailed implementation factors, TLEF evaluates three high-level user-oriented metrics, $\langle cr, ws, rs\rangle$. Under the specific configuration $conf$, we get the compression rate (short as cr), average writing speed (short as ws), average reading speed (short as rs). As the data may be stored in different node as the node of computing task, rs consists of the local reading speed rs_{local} and remote reading speed rs_{remote}.

4.2 Framework Architecture

The architecture of TLEF is demonstrated in Fig.3. Users can input the workloads (expressed as HiveQL language), table layout configuration to be override and table specifications (including the schema of tables and other configurations

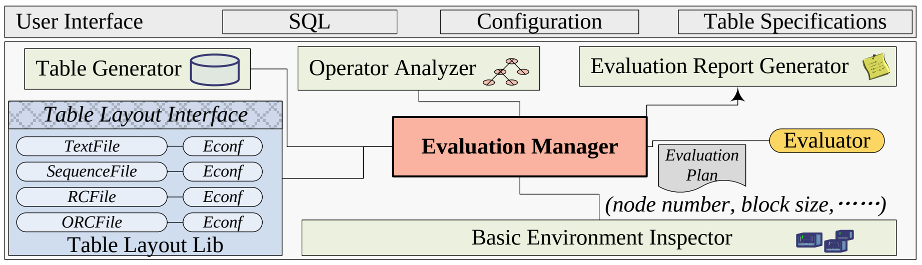


Fig. 3. The Architecture of Table Layouts Evaluation Framework.

like the range of a string column's length) through TLEF's user interface. The core of TLEF is the evaluation manager, which generates the evaluation plan and calls the evaluator to generate the read/write workloads.

The *Basic Environment Inspector* inspects the environment-specific configurations, such as the cluster's node number, the block size in HDFS and so on. The *Operator Analyzer* parses the query and extracts the table-layout-related operations (including projection and predicates) that can be pushed down to the storage layer. For the general cases, the *Table Generator* randomly generates the tables according to the table specifications. The *Table Layout Lib* not only contains built-in table layouts, but also exposes the unified interfaces. Developers can implement the interfaces for the upcoming new table layouts. The *Result Reporter* produces the evaluation metrics, based on which TLEF will make suggestions on table layout.

4.3 Configuration Space Exploration

There are a large number of variables that can be specified by users in table layout configuration and the default configuration may not be expected to be the most efficient. A straightforward exploration approach is to apply enumeration and search techniques to the full space of parameter settings. Some parameters have small and finite domains, e.g., compression type, while the domain of some parameters is unbounded. In TLEF, each implemented table layout enumerates the domains in *Econf* for individual configuration parameters. The exploration of our evaluation framework is as follows: (1) Evaluate individual parameters. For each parameter, set other parameters as default value, test the table layout with the parameter setting in *Econf* to get the metrics. (2) Heuristic search with user defined algorithm. TLEF exposes the algorithm interface for user.

4.4 Evaluation Report: Extending to Distributed Environment

Instead of solely making the choice, TLEF generates the evaluation report to users, including the contents of three respects: the high level user-oriented metrics, table layout suggestion for each query and the comprehensive analysis.

Assume the table has n rows, the row size can be estimated as rsz according to the schema; then the table size is $ts = cr \times n \times rsz$. We note the MapReduce split size as *splitsize* and the total mapper slots number as msn. Then the length of mapper round can be calculated as $\lceil ts/(splitszie \times msn) \rceil$. For example, there are $\lceil 1204/(64 \times 16) \rceil = 2$ rounds of mappers for a 1GB table with the configuration of 64MB split size and 12 mapper slots.

Table layout suggestion for each query. TLEF lists the table-layout-related operations and calculates the new average read speed under column projection, $rs_{new} = rs \times projectedSize/rowSize$. During the query execution, the results produced by each reducer are certain and irrelevant to the table layouts; hence the write performance is determined by the metric ws. For example, when the query is data loading, table layout with better ws is preferred. As for the read performance, TLEF suggests the candidate with efficient $rs_{new} \times \lceil ts/(splitszie \times msn) \rceil$. In the distributed environment, the IO read performance may be disturbed by many factors. For the local and remote cases, we respectively define the perturbation functions ω_l and ω_r. Then the upper bound and lower bound can be estimated as the following expressions. $\lceil ts/(splitszie \times msn) \rceil \times \omega_l(rs_{local})$ and $\lceil ts/(splitszie \times msn) \rceil \times \omega_r(rs_{remote})$.

Comprehensive analysis. It is straightforward for users to make choice according to the table layout suggestions for each query. Generally, there is a tradeoff of storage space utilization and query performance, which depend on user's preference. If the previous case is the main consideration, table layout with efficient cr metric is preferred. For the latter one, it depends on different queries' weights (e.g., the one with the largest proportion or the one for the most frequent query, and etc.); we discuss some cases here. When the workload only includes data loading, table layout with better ws is preferred. However, *write-once-read-more* is frequent and query performance is usually the focus in MapReduce based query systems. For the extreme case of no extracted operation, a row-oriented table layout is appropriate for all column scans. For the general case with many table-layout-related operations, a table layout supporting them is important.

5 Experimental Study

In this section, we conduct experimental study from two dimensions. First, we empirically evaluate the popular table layouts from the perspective of a single task; we then extend the results to the distributed environment. Next we make discussion on the intermediate table layout strategy for query optimization.

Out experimental study is conducted on a local cluster of 10 DELL OptiPlex-990 nodes connected by a 1GB ethernet switch. Each node is equipped with four Intel i7-2600 3.4GHz cores, 16GB RAM and 2TB hard disk drives. Operating system is Ubuntu-11.04 x86_64. Hadoop 1.2.0 and Hive 0.12.0 are used. One node is reserved for Hadoop JobTracker and the NameNode. The other 9 nodes are used for HDFS DataNode and MapReduce TaskTracker. The HDFS block size is 256MB and each file has 2 replications. In fact, these settings can be automatically detected by basic environment inspector.

5.1 Empirical Performance Analysis

We use a synthetic relational table generated by TLEF table generator as follows: each record consists of an incremental integer ID, 6 string columns and 6 integer columns. The integers are randomly assigned values between 0 and 100000; random strings of length between 10 and 40 are generated over readable ASCII characters, similar to the schema defined in [11].

Metrics with Default Configuration. In practice, table layouts are often adopted with the default configuration, especially for the none-expert users. As the default configuration indicates the most common usage case, we first conduct experiments to study the general performance for table layouts. Table sizes are respectively 242.5MB, 263.9MB, 184.2MB and 169.7MB for TextFile, SequenceFile, RCFile and ORCFile. We write and read whole column set without the upper layer's query semantics and the results are shown in Fig.4. It can be seen that, fine-grained column encoding schemes gain significant compression rate and row-oriented table layouts, without extra metadata maintenance and row reconstruction, outperform others for data loading and row scanning.

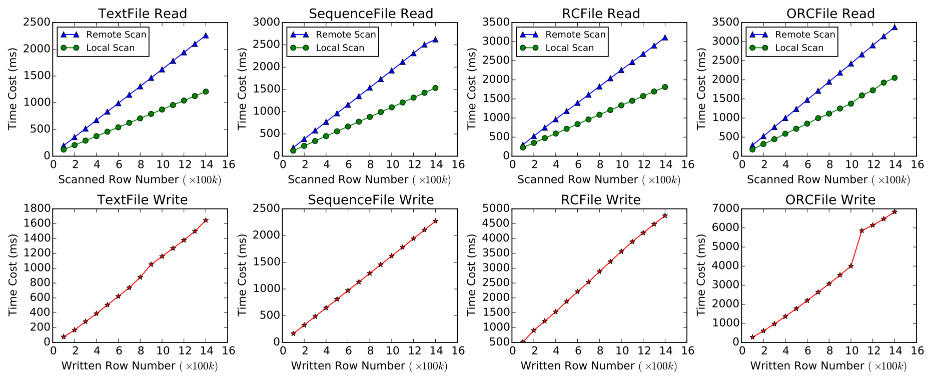


Fig. 4. Table Layout IO Performance with Default Configuration.

Configuration Space Exploration. We take two configuration parameters for ORCFile, *compression-kind* with the finite domain {None, Zlib, Snappy} and the *strip-size* with the infinite domain, to show the exploration process. The strip size varies with the values in geometrical sequences {256MB, 64MB, 16MB, 4MB, 1MB}, which is automatically generated by TLEF according to the properties defined in table layout lib. As depicted in Fig.5, the evaluated metrics varies according to different configurations. But these differences are not very significant, diversed in the same order of magnitude.

Query Aware. As only ORCFile supports both column projection and predicate pushdown, we take it as the example to demonstrate how the query semantics can be leveraged. The client respectively reads 1400000 rows in the local mode with 2 integer column, 1 integer column and 1 string column, and all

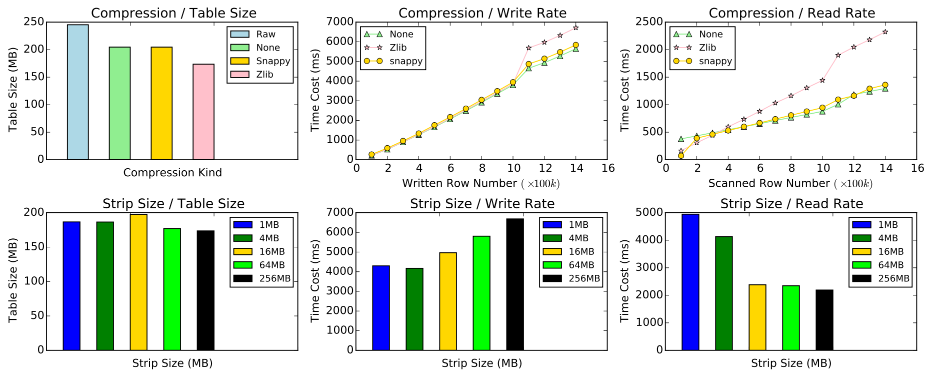


Fig. 5. Configuration Space Exploration Example: Compression and Strip Size.

columns. The left and middle graphics in Fig.6 respectively show the read time costs and the data sizes in different cases. The results indicate the effectiveness of column projection for filtering unnecessary columns.

For the predicate pushdown, ORCFile will check the statistics every configured number and the rows will be skipped if it doesn't satisfied the predicates. As the default *index-strip* is 10000 (maintaining column statistics, like max and min, every 10000 rows to skip rows), we evaluate the predicates $\sigma_1 =' id < 140000'$, $\sigma_2 =' 140000 < id < 830000'$ and $\sigma_3 =' id > 139999'$. It can be seen from the right graphic of Fig.6 that the effectiveness of predicate push down is heavily relied on the predicate's selectivity on data.

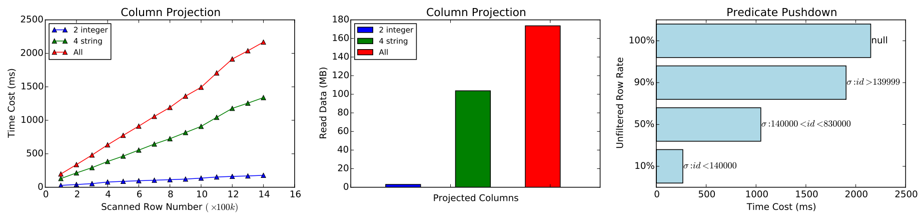


Fig. 6. Column Projection and Predicates Pushdown.

Distributed Environment. We generate the table with 56 millions rows (about 9.2GB) stored as textfile. The benchmark query is *"select count(*) from T"*, which will be translated into one MapReduce job with 39 mappers and 1 reducer in our environment. Fig.7 demonstrates the runtime of query's execution. It can be seen that the performance of task with local data is stable, while those with the remote data differ greatly.

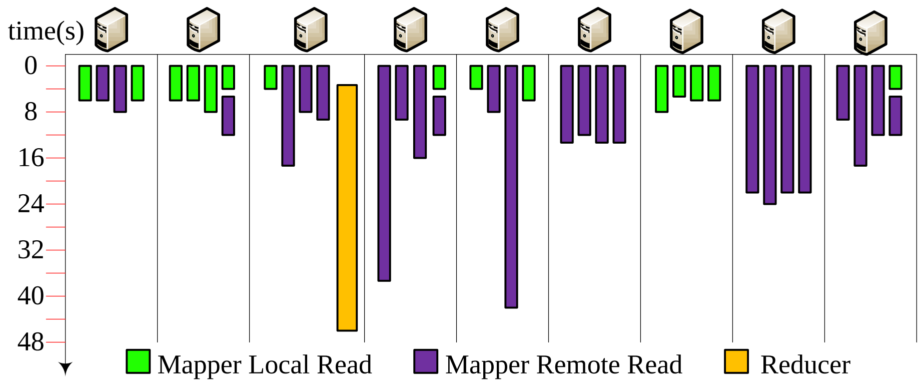


Fig. 7. Table Layout IO Performance in Distributed Environment.

5.2 Intermediate Table Layout for High Level Query Systems

Assume the producer query Q_1 has n columns $WC = \{WC_i | 1 \leq i \leq n\}$ and the consumer query Q_2 will read m columns $RC = \{RC_i | 1 \leq i \leq m\}$. To investigate whether the intermediate table layout can be adaptively set based on the Q_1 and Q_2, we discuss the characters of column projection and predicate push down.

Column projection. Generally, the column pruning (called "cp") is adopted in high level language systems' query optimizers. With this optimization, Q_2 reads all columns produced by Q_1, i.e., $m = n$ and $WC = RC$. Hence, Table layouts that support column projection can not exploit the advantage.

Predicate push down. Similarly, predicate push down (called "ppd") is also supported in query optimizers. Assume the predicate is for one column WC_k in Q_2, if and only if the WC_k is a dynamically new column generated by Q_1 (i.e., the aggregated column, calculated by user-defined functions and etc.), the predicate push down can be exploited. For instance, Q_1 is *"select col1, sum(c2) as col2 group by col1"* and Q_2 is *"select co1, col2 where col2>2"*.

In addition, the intermediate table size can only be determined after the execution of Q_1. Optimizing the intermediate table layout consequently brings not too much improvement on performance. Considering Q_2 scans all columns, setting the intermediate table layout as a row-oriented one (for example, the default *sequencefile* setting in Hive) is reasonable for most cases.

6 Conclusion

In this paper, we propose the testing approach based on black box method and the query aware strategy to evaluate the table layouts in MapReduce based environment. To assist users make the appropriate choice under specific workloads, we develop a lightweight evaluation framework. It automatically extracts table-layout-related operations from queries and rapidly evaluates the metrics.

Acknowledgement

This work was partially supported by National Science Technology Support Plan (2013BAH05F03) and National Natural Sciences Foundation of China (61202065, U1435220, and 61170074).

References

1. Apache Hadoop. http://hadoop.apache.org/.
2. ORCFile. https://issues.apache.org/jira/browse/HIVE-3874.
3. Parquet. A Columnar Storage Format for Hadoop. http://parquet.io/.
4. Trevni. http://avro.apache.org/docs/1.7.6/trevni/spec.html.
5. Zebra. Columnar Storage Format. https://wiki.apache.org/pig/zebra.
6. J Dean, S Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. in OSDI Conference, 2004.
7. Thusoo A, Sarma J S, Jain N, et al. Hive-a Petabyte Scale Data Warehouse using Hadoop. in ICDE Conference, 2010.
8. Olston C, Reed B, Srivastava U, et al. Pig Latin: a Not So Foreign Language for Data Processing. SIGMOD, 2008.
9. Y. He, et al. RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems. ICDE, 2011.
10. Lin, et al. Llama: Leveraging Columnar Storage for Scalable Join Processing in the MapReduce Framework. SIGMOD, 2011.
11. Floratou Avrilia, Patel J M, Shekita E J, et al. Column-Oriented Storage Techniques for MapReduce. VLDB, 2011.
12. Guo S, Xiong J, Wang W, et al. Mastiff: A MapReduce-based System for Time-Based Big Data Analytics. CLUSTER, 2012.
13. Jindal Alekh, Jorge-Arnulfo Quiane-Ruiz, et al. Trojan Data Layouts: Right Shoes for a Running Elephant. SOCC, 2011.
14. Yin Huai, et al. Understanding Insights into the Basic Structure and Essential Issues of Table Placement Methods in Clusters. VLDB, 2014.
15. R. Ramakrishnan, et al. Database Management Systems, McGraw-Hill, 2003.
16. G. P. Copeland and S. Khoshafian, A Decomposition Storage Model, in SIGMOD Conference, 1985, pp. 268-279.
17. A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, et al. C-store: A Column-oriented DBMS. VLDB, 2005.
18. D.J. Abadi, S. Madden, and N. Hachem, Column-stores vs. Row-stores: How Different are They Really? SIGMOD, 2008.
19. D. Tsirogiannis, S. Harizopoulos, M. A. Shah, rt al. Query Processing Techniques for Solid State Drives. SIGMOD, 2009.
20. H. Herodotou and S. Babu. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. VLDB, 2011.
21. Nykiel T, Potamias M, Mishra C, Kollios G, et al. MRShare: Sharing Across Multiple Queries in MapReduce. VLDB, 2010.
22. Lim, Harold, Herodotos Herodotou, et al. Stubby: a Transformation-based Optimizer for MapReduce Workflows. VLDB, 2012.
23. Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, et al. Ysmart: Yet Another Sql to Mapreduce Translator. ICDCS, 2011.