2018 IEEE 34th International Conference on Data Engineering

Rainbow: Adaptive Layout Optimization for Wide Tables

Haoqiong Bian, Youxian Tao, Guodong Jin, Yueguo Chen(✉), Xiongpai Qin, Xiaoyong Du

School of Information, Renmin University of China, Beijing, China
DEKE Key Laboratory (Renmin University of China), MOE, Beijing, China
{bianhq,taoyouxian,jinguodong,chenyueguo,qxp1990,duyong}@ruc.edu.cn

***Abstract*—Popular column stores such as ORC and Parquet have been widely used in many Hadoop-oriented data analysis systems. With the effective column skipping and data compression functionalities provided by column stores, wide tables with hundreds or even thousands of columns are applied by many big data analysis applications to avoid the expensive distributed joins. We found that the performance of such systems can be further improved by optimizing the physical data layout to fit certain workloads and system settings. However, it is nontrivial to perform such optimization manually.**

In this demo, we present a data layout optimization tool called Rainbow, which leverages workload-driven layout optimization algorithms to adjust data layouts adaptively without intervening the previous data blocks that have been stored. We also provide a Web UI for users to interact with the layout optimization process. Furthermore, Rainbow is open sourced with an accompanying benchmark for performance evaluation of wide tables.

I. Introduction

Nowadays, many big data analysis systems share HDFS (Hadoop Distributed File System) as their common underlying storage [1], [2]. Large amount of data from various sources continuously converges and is further dumped into HDFS. As the de-facto standard of distributed data storage, HDFS provides big data systems an unified data storage with merits of fault-tolerance, massive scalability and high R/W throughputs.

Many big data analysis systems such as Hive [1] and Spark [2] have been built to support both interactive and batch analysis of the huge and ever-increasing data in HDFS. In many cases, they present data as two-dimensional wide tables with the number of columns from a few hundreds to even thousands. These wide tables are commonly stored as columnar files such as RCFile [3], ORC [4], Parquet [5] and CarbonData [6], because of but not limited to the following advantages: 1) all the required columns of an analytical task are likely to be read from one single table so that the expensive join cost can be saved; 2) new columns can be easily added to an existing table without affecting the existing data analysis applications; 3) only required columns of a query are fetched without reading extra data; 4) data is well compressed by the efficient encoding and compression algorithms.

Although wide table provides very good performance and the support of application-friendly schema evolution, we found that the data reading latency of systems running on top of wide tables can be further reduced with column ordering [7]. By column ordering, we reorder the physical storage of columns inside a row group – a horizontal partition of a table. As a result, frequently co-accessed columns are put closer in a row group, so that the seek latency – the root cause of data reading latency – can be largely reduced. Column order is transparent to queries because of the self-describing mechanisms applied in individual row groups. It therefore allows a table to be comprised of a set of row groups with different column orders.

Besides column order, row group size (RGS, the bytes in a row group) is proven to be another important factor that affects the reading performance of wide tables [7], [8]. Larger RGS leads to less row groups and thus less total number of disk seeks. Since the seek cost is sublinear to the seek distance [7], larger RGS will generate lower total seek cost for reading the same amount of data. Furthermore, if the RGS is too small, a reading task (often reads a few columns from a row group) is likely to read only a small amount of data, leading to the task scheduling and initialization overheads to be the major performance bottleneck. However, a larger row group consumes more memory during query processing, and may reduce the degree of data processing parallelism or even cause OOM errors. It is therefore not that easy to set a proper RGS for various workloads and system settings manually.

In this demonstration, we present an adaptive wide table layout optimization tool called Rainbow. It provides a Web user interface in which user can create a pipeline to transform and load data from a specific data source into a wide table in HDFS. Data is fetched from the data source, packed into mini batches, and loaded into the wide table per mini batch. As the query workload being collected, Rainbow automatically triggers a layout optimization procedure to set a proper RGS and derive an approximation of the optimal column order (as the column ordering problem is NP-Hard [7]). Estimated performance gain of the optimized layout is shown in a scatter diagram. Users can choose to apply the optimized layout or further evaluate the performance gain in a real HDFS cluster before making the decision. The new layout will be applied by Rainbow in the next mini batch.

Rainbow also provides a command line interface, which is more convenient for data layout optimization of static data and workloads. Rainbow is open sourced [1] with an accompanying benchmark for wide tables. The modules in Rainbow are designed to be a set of loose coupling Java libraries, which can be easily embedded in other ETL or data analysis systems.

[1] https://github.com/dbiir/rainbow

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00200

II. System Overview

As shown in Fig. 1, Rainbow contains six modules (in gray). The *Web UI* module and the *CLI* (command line interface) module are user interfaces, and the other modules are designed as a set of libraries. Rainbow collects the workload information such as the accessed columns of queries, without issuing the queries for users. This allows us to build Rainbow as an external module without modifying the existing data analysis applications and systems. Details of the Rainbow's modules are discussed as follows.

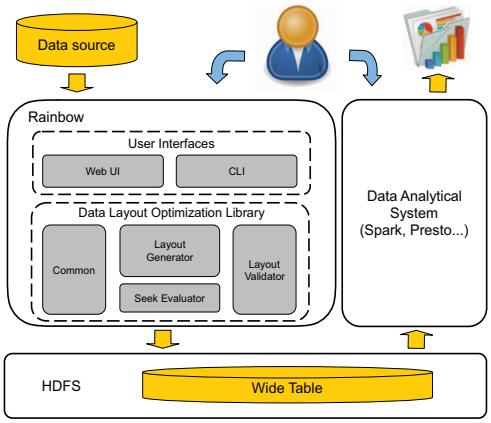

Fig. 1. System architecture of Rainbow

A. Web UI, CLI, and Common

The *Web UI* module works in a client-server mode. The front-end provides a user-friendly Web interface while the back-end invokes underlying modules. With the Web UI, users can interact with Rainbow to perform ETL and optimize the data layout adaptively. RESTful APIs are provided in this module to collect the workload information from applications for workload-driven layout optimization.

The *CLI* provides a set of commands which are convenient for layout optimization of static workloads. Users can easily deploy and run the CLI module anywhere. The *Common* module contains the shared functions and interfaces in Rainbow. It maintains the configurations and runtime logs as well.

B. Layout Generator

The *Layout Generator* implements the data layout optimization algorithms proposed in [7]. Our previous solution [7] only considers seek cost in the cost model and optimizes the column order accordingly. In Rainbow, we revise the cost model so that the RGS factor is also considered.

To provide a better user experience and convince the user that an optimized data layout is 'good' enough, *Layout Generator* provides the estimated query performance on both the baseline layout and the optimized layout. Based on the estimation, a scatter diagram is provided in the Web UI.

C. Layout Validator

In some production environments such as a multi-tenant system, performance guarantee is often required by some high-priority queries. Although the weight of each query is considered in our layout optimization algorithms [7] so that it is not likely to punish an 'important' query, user may still want to test the query performance in real environment to make sure that the optimized layout is good enough. *Layout Validator* helps us to do that. It issues queries to a Spark or Presto (currently supported) and measures the end-to-end elapsing time of each query. A small validation cluster can be configured for the performance validation so that the production cluster will not be affected.

D. Seek Evaluator

This component is responsible for deriving the seek cost function according to the underlying hardware and file system. The seek cost function is used to establish the cost model in Rainbow. Details of seek cost evaluation is discussed in [7].

III. Key Techniques

A. Cost Model

In Rainbow, we focus on reducing data reading cost, which mainly comes from the first map stage of query execution in a MapReduce-like system, and takes a majority of the end-to-end latency for many batch queries [7]. Basically, the reading cost of a map task (we assume one row group is read by a map task, although the model can be easily adapted to support multiple row groups) includes three major parts:

- *Constant overhead*. It includes the cost of task scheduling, metadata parsing, garbage collection, and the initial seek cost to read a row group.
- *Sequential reading cost*. Given the access pattern $AP = \{c_{q,1}, c_{q,2}, \ldots, c_{q,m}\}$ of a query q, the sequential reading cost on a row group is $SeqRead(q) = \sum_{i=1}^{m} size(c_{q,i})/b$, where b is the sequential read bandwidth of the disk, $c_{q,i}$ is the i^{th} column that is required by q.
- *Seek cost*. Given a column order $S = \{c_1, c_2, \ldots, c_n\}$, the column access pattern $AP = \{c_{q,1}, c_{q,2}, \ldots, c_{q,m}\}$ of query q, the seek cost of a row group is $Seek(q) = \sum_{i=1}^{m-1} f(dist(c_{q,i}, c_{q,i+1}))$, where $dist$ is the distance in bytes between two data items in a file, and f is the seek cost function built by *Seek Evaluator* in Section II-D.

Definition 1 (Query reading cost): Given a query q, a wide table of N row groups, the reading cost of q is

$$Cost(q) = N \times (\epsilon + SeqRead(q) + Seek(q)) \quad (1)$$

where ϵ is the constant overhead of reading a row group. We assume that the reading cost of each row group is the same. The reading cost of the whole workload is then modeled as:

Definition 2 (Workload reading cost): Given a weight w_q for each query $q \in Q$, the reading cost of a workload Q is

$$Cost(Q) = \sum_{q \in Q} (w_q \times Cost(q)) \quad (2)$$

The weight w_q implies the frequency/importance of the query q. We apply the workload reading cost as the target to be optimized for the layout optimization component.

B. Column Ordering

Given a seek cost function and a workload, the column ordering problem [7] is to find an optimal column order which brings the minimal seek cost to the workload. This is proved to be NP-Hard [7], so that we propose a simulated-annealing based column ordering algorithm called *SCOA* in [7] to solve this problem. In *SCOA*, a column order is a state, and the seek cost is the energy of a state. Two randomly selected columns in the current state are swapped to generate a neighbour state. By accepting the neighbour state probabilistically in a loop following the annealing schedule, *SCOA* converges to an approximation of the optimal column order.

C. Setting Row Group Size

Row group size (RGS) affects the data reading cost of queries in two aspects:

- RGS affects the size of each column and the distance between columns in a row group, which further affects the seek cost of queries.
- Given a table of certain size, the RGS affects the number of row groups N in the table, which further affects the data reading cost of queries.

We can however find that the larger RGS is, the lower data reading cost will be. Accordingly, in Rainbow, we set RGS according to the memory limitation. Given the allocated memory size M and parallel degree P of reading tasks configured on each node in the production cluster, the best RGS set by Rainbow is $RGS_{best} = M/P/a$, where a is the memory amplifying factor of a reading task. It is calculated by dividing the maximum memory consumption of a row group reading task by RGS.

D. Workload Evolution

Workload-driven layout optimization is based on the assumption that the workload evolves gradually without mutation, so that we can predict how data will be accessed from the recent workload. An effective solution for workload evolution, which timely detects and discards the outdating queries, is necessary for workload-driven layout optimization.

In Rainbow, we propose a solution based on two assumptions: 1) the temporal locality assumption that the same query is likely to be re-executed within a certain time duration. 2) the frequency assumption that frequently executed queries are more likely to be re-executed in the coming future.

As the layout is optimized according to the access patterns (APs) of queries, in our solution, we maintain a cache of APs (denoted as APC). New APs are inserted into APC and the outdating APs are evicted. A LRU-based caching policy in Algorithm 1 is proposed to perform the workload evolution process. When a new query is collected by Rainbow, the AP of the query is passed to the algorithm. The data structure APC maintains the latest timestamp and the weight of each AP. APs in APC are ordered by the timestamp. The weight of an AP is initialized as 1 and is incremented each time the same AP hits APC (line 4). Weights of the APs will be used in layout optimization so that the performance of the APs with higher weights will be optimized preferentially.

Algorithm 1: AccessPatternCaching

```
   Input: The access pattern AP = {c_{q,1}, c_{q,2}, ..., c_{q,m}}
          of a query q
1  t := current time;
2  if APC.hits(AP) then
3      APC.updateTimestamp(AP, t);
4      APC.incrementWeight(AP);
5  else
6      APC.insert(AP, t);
7      updateCounter++;
8      AP_e := APC.earliestAP();
9      if AP_e.timestamp < t − L then
10         if prevSize == 0 then
11             updateCounter = 0;
12             prevSize = APC.size;
13             trigger layout optimization and return;
14         APC.evict(AP_e);
15         updateCounter++;
16 if prevSize > 0 and updateCounter/prevSize > θ
     then
17     updateCounter = 0;
18     prevSize = APC.size;
19     trigger layout optimization;
```

An AP lifetime L is set by the user for an ETL/optimization pipeline. It is used to evict the AP with the earliest timestamp in APC earlier than $t - L$ (line 9-14). The cache size is controlled by L. This is more intuitive and effective than setting a fixed cache size in [9] and [10]. The variable $prevSize$ records the size (number of elements) of APC at the time of the last layout optimization. It is initialized as 0 when the system starts. The layout optimization is firstly triggered when the first eviction happens. A threshold θ is also configured by the user. It is the percentage of elements updated since the last layout optimization. After the first layout optimization, in the case of $updateCounter/prevSize > \theta$ (line 16), a new layout optimization process will be triggered to make the layout adaptive to the updating workload.

IV. Demonstration

In this section, we present how Rainbow is used in layout optimization. A dataset of 500 GB and 1000 columns generated by the accompanying benchmark will be used in demonstration. The benchmark generates data and workload following the data template and workload template created according to the real-world use cases of wide table analysis in Microsoft Bing [7]. A 5-node (1 master and 4 slaves) production cluster and a 3-node validation cluster (1 master+slave and 2 slaves, with Rainbow running on the master) are used in this demonstration.

A. Pipeline Creation

Fig. 2 shows the Web UI of Rainbow. On the top navigation bar, user can switch to the *Pipeline* tab for creating a pipeline. To create a pipeline, users need to fill in some settings in a pop-up form, which includes: 1) The type and URL of data source. Different data sources such as HDFS and Kafka can be plugged into Rainbow; 2) Initial RGS and the data schema (without layout optimization); 3) the target storage path of the wide table on HDFS, into which the loaded data will be stored; 4) the lifetime of the query access patterns and the workload evolution threshold (discussed in Section III-D).

In this demo, a pipeline will be running in a streaming manner – source data is read per mini batch (4 GB by default), and then transformed (by MapReduce) and loaded into HDFS.

B. Layout Optimization

While the ETL pipeline is running, data layout can be optimized on the fly. As shown in Fig. 2, users can interact with the layout optimization process of a pipeline on the *Optimization* tab page. The whole layout optimization process includes several steps, which are divided into separate panels on the page.

Firstly, a small data sample (10 GB by default) is copied from data source to the validation cluster. This sample is then transformed into columnar format to evaluate the data compression ratio, the size of columns in a row group and the memory amplifying factor. For workload, the access patterns of queries are submitted to the pipeline through the RESTful API. While the workload being submitted to the pipeline, the layout optimization is triggered adaptively by the workload evolution solution.

Once an optimized layout is derived, its estimated performance gain against the previous applied layout (or the initial layout) will be then shown intuitively through a scatter diagram in the *Layout Strategy* panel. User can interact with the scatter diagram to see detailed information of each query.

User can apply the optimized layout in the next mini batch of ETL by clicking the *Accept* button in the *Layout Strategy* panel, or further validate the performance gain in a scatter diagram in the *validation* panel by clicking the *Validate* button. The optimized layout will be discarded if it is not accepted before the next layout is derived. A timeline is also shown in the bottom of the page, in which user can find the historical actions performed on the pipeline.

Demonstration videos and documents of Rainbow are provided in our git repository (https://github.com/dbiir/rainbow).

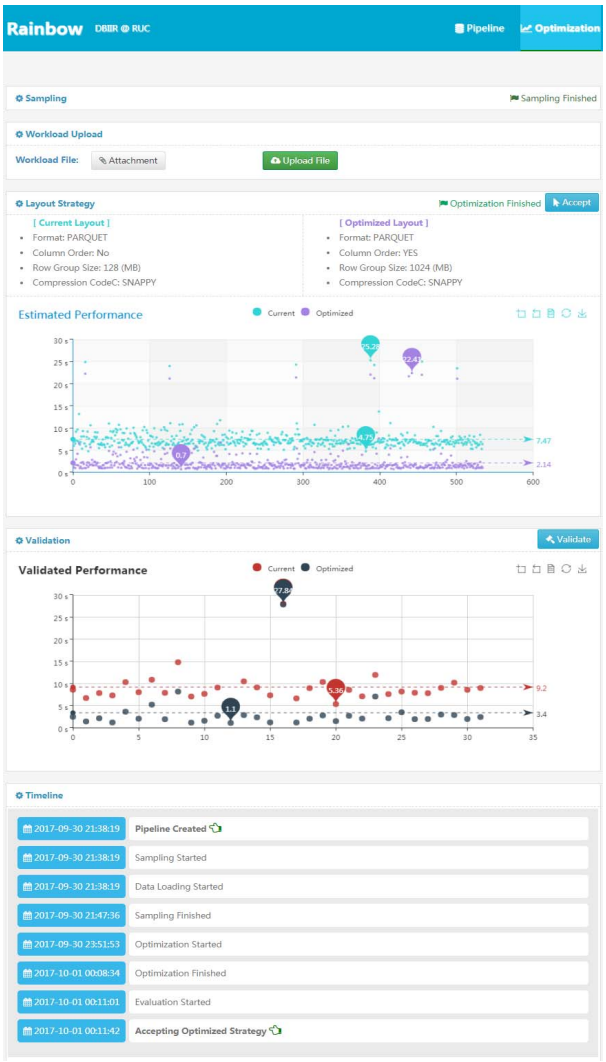

Fig. 2. The Web user interface of layout optimization

V. Acknowledgements

Yueguo Chen is the corresponding author. This work is supported by Science and Technology Planning Project of Guangdong under grant (No. 2015B010131015), the National Science Foundation of China under grant (No. U1711261, 61472426 and 61432006), and the open research program of State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Science (No. CARCH201510).

References

[1] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy, "Hive - a petabyte scale data warehouse using hadoop," in *ICDE*, 2010, pp. 996–1005.

[2] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark sql: Relational data processing in spark," in *SIGMOD*, 2015, pp. 1383–1394.

[3] 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, pp. 1199–1208.

[4] Orc. [Online]. Available: https://orc.apache.org/

[5] Parquet. [Online]. Available: http://parquet.apache.org/

[6] Carbondata. [Online]. Available: http://carbondata.apache.org/

[7] H. Bian, Y. Yan, W. Tao, L. J. Chen, Y. Chen, X. Du, and T. Moscibroda, "Wide table layout optimization based on column ordering and duplication," in *SIGMOD*, 2017, pp. 299–314.

[8] Y. Huai, S. Ma, R. Lee, O. O'Malley, and X. Zhang, "Understanding insights into the basic structure and essential issues of table placement methods in clusters," *PVLDB*, vol. 6, no. 14, pp. 1750–1761, 2013.

[9] I. Alagiannis, S. Idreos, and A. Ailamaki, "H2o: a hands-free adaptive store," in *SIGMOD*, 2014.

[10] J. Arulraj, A. Pavlo, and P. Menon, "Bridging the archipelago between row-stores and column-stores for hybrid workloads," in *SIGMOD*, 2016, pp. 583–598.