Wide Table Layout Optimization based on Column Ordering and Duplication

Haoqiong Bian[1], Ying Yan[2*], Wenbo Tao[3], Liang Jeff Chen[2], Yueguo Chen[1*],
Xiaoyong Du[1], Thomas Moscibroda[2]
[1]DEKE Key Lab (MOE), Renmin University of China; [2]Microsoft Research; [3]MIT
[1]{bianhq, chenyueguo, duyong}@ruc.edu.cn,
[2]{ying.yan, jeche, moscitho}@microsoft.com, [3]wenbo@mit.edu

ABSTRACT

Modern data analytical tasks often witness very wide tables, from a few hundred columns to a few thousand. While it is commonly agreed that column stores are an appropriate data format for wide tables and analytical workloads, the physical *order of columns* has not been investigated. Column ordering plays a critical role in I/O performance, because in wide tables accessing the columns in a single horizontal partition may involve multiple disk seeks. An optimal column ordering will incur minimal cumulative disk seek costs for the set of queries applied to the data. In this paper, we aim to find such an optimal column layout to maximize I/O performance. Specifically, we study two problems for column stores on HDFS: column ordering and column duplication. Column ordering seeks an approximately optimal order of columns; column duplication complements column ordering in that some columns may be duplicated multiple times to reduce contention among the queries' diverse requirements on the column order. We consider an actual fine-grained cost model for column accesses and propose algorithms that take a query workload as input and output a column ordering strategy with or without storage redundancy that significantly improves the overall I/O performance. Experimental results over real-life data and production query workloads confirm the effectiveness of the proposed algorithms in diverse settings.

1. INTRODUCTION

The challenge of Big Data has shifted the design of data analytical systems from single machines to large-scale distributed systems. While there are many distributed data analytical systems in the market and their runtime specialties vary greatly, they all share a common core as their underlying storage engine: HDFS (Hadoop Distributed File System). In HDFS, a common data modeling is to represent data as two-dimensional tables. A table is horizontally partitioned to scale out and to leverage multi-machine parallelism. At runtime, each table partition, a.k.a. a *row group*, is read and processed individually by mappers in each machine.

The physical layout of a row group plays a fundamental and critical role in system I/O performance [27, 34, 29, 30]. Existing studies have focused on two aspects in organizing data in a row group: i) column store vs. row store and ii) row group size. A *row-oriented store* serializes a row as a blob and allows an application to retrieve one row at a time. While this data format is common in conventional databases, modern data analytical systems on HDFS have primarily opted for column stores, e.g., RCFile [29], ORC File [5] and Parquet [3]. A *column store* serializes a whole column in a row group as a blob. This is good for I/O performance for two reasons: First, many queries only access a small number of columns, even though the underlying table is very wide; second, storing a column in its entirety facilitates compression. It has been shown that when the row group size is sufficiently large, the additional cost incurred by row reconstruction can be amortized and is eventually almost negligible [30]. In practice, the row group size is set to a value that ensures massively parallel data processing while still keeping re-do cost reasonable in case of a mapper/reader failure.

In this paper, we study a new dimension of organizing data in a row group: *column ordering*. Column ordering in a column store specifies how columns are physically ordered so that two adjacent columns can be accessed with sequential reads. Column ordering is not an important issue when a table only has dozens of columns. However, in Bing Microsoft, data analysis pipelines commonly feature very wide tables with thousands of columns. These tables are either raw logs or products of cooking logs to facilitate other analysis jobs down the pipeline. Very wide tables are common in modern data analytical systems, because of their advantages in analytical processing compared to normalizing/partitioning data into less wide tables (e.g. [36]). In one specific instance, an in-production table contains 1187 columns and grows at a pace of 5TB per day. For such a wide table, finding the right column order can lead to dramatic differences in performance.

To understand the benefits of column ordering, we select three frequently-executed queries from Bing Microsoft's data analytical pipeline, namely A, B and C, following the query template

```
SELECT $col_{i_1}, col_{i_2}, \ldots$
FROM TargetTable
[WHERE cond]
[GROUP BY $col_{j_1}, col_{j_2}, \ldots$]
[ORDER BY $col_{k_1}, col_{k_2}, \ldots$]
```

and measure their performance on a column store with different column orders. We execute the queries on the default 1187-column table (unordered) in the actual production environment, as well as on its duplicate whose columns are manually ordered so that columns accessed by the queries are physically close. The results show that column ordering reduces end-to-end query execution time by up to 70%. These results are understandable. In this workload, disk I/O is the main cost of query execution and disk seek is a dominant factor in disk I/O. If columns accessed by a query are physically placed

[*] Yueguo Chen and Ying Yan are the corresponding authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD'17, May 14-19, 2017, Chicago, Illinois, USA.
© 2017 ACM. ISBN 978-1-4503-4197-4/17/05...$15.00
DOI: http://dx.doi.org/10.1145/3035918.3035930

together, disk I/O only involves one seek. By comparison, when a table is very wide and accessed columns are far apart, the number of disk seeks of executing one query increases, significantly bogging down I/O performance.

Choosing an appropriate *column order* is thus critical in improving I/O performance for queries on very wide tables. However, the solution is non-trivial given that there are thousands of columns and the queries accessing them have complex patterns. For instance, the query workload for the above wide table contains thousands of queries, each accessing from a few to several hundreds of different columns. Organizing the columns to efficiently accommodate one query may negatively impact the performance of many other queries. We thus seek an algorithmic solution to choose a column order that optimizes the overall performance across all queries.

While a column-ordering algorithm tries to strike a balance among different queries, contention is still unavoidable in general. This challenge can be tackled by *column duplication*. Storage capacity is usually the least concerned factor in the overall cost of a large-scale data processing system. Column duplication takes advantage of storage redundancy and duplicates certain columns, placing each copy in a different position to ease the contention among queries. While causing space overhead, this technique is effective in reducing disk seek costs. Column duplication adds an additional dimension to the problem of column ordering, and hence demands new algorithms that co-optimize column ordering and duplication.

In this paper, we show that column ordering plays a key role in data analytics performance on wide tables. Specifically, we make the following contributions:

- We propose an I/O cost model for column stores in HDFS based on an empirical performance study of modern disks. We then formalize the column ordering problem with optional column duplication so that the I/O cost of a query workload is minimized.

- We propose an algorithm, namely SCOA (Simulated Annealing based column ordering algorithm), that efficiently finds an approximation of the optimal column order. We further optimize I/O throughput by combining SCOA with a storage-constrained column duplication algorithm.

- We use in-production data and query workloads and perform an experimental study on the cost model and the proposed algorithms. The results show that our solutions achieve significant gains in all tested settings, in some cases by up to 73.9% I/O performance gain. We can get additional 12% performance gain by column duplication with less than 5% storage headroom.

2. PRELIMINARIES

In this section, we first review the design of hard disks and give a formal definition of disk seek cost. We then review data layout in today's HDFS-based column stores and formalize the problem of column ordering.

2.1 Disk Seek Cost

Hard disk drives (HDDs) are the most common media for storing a large volume of data in HDFS, due to their large capacity, low price per storage unit and much better durability. In HDD, seek cost is the main performance bottleneck when data is not read sequentially [22]. Disk seek cost is mostly considered as a constant value (i.e., the average seek cost) in the literature [16, 45, 15]. In this paper, we use a simple yet accurate model (Appendix A) to quantify disk seek cost for optimizing big wide table layout.

In file systems, data is stored in files, which are conceptually byte arrays. A data reader inputs the starting offset and the length in bytes to read a chunck of data. Therefore, we model the seek cost as a function of the seek distance in bytes. More formally, given a data object i (e.g., a column), let $s(i)$ be the size of the data object i (number of bytes that i occupies). Assume that a data object is sequentially written to a file through the OS API. Let $b(i)$ be the offset of the first byte of the data object i in the file; and $e(i)$ be the offset of the byte next to the last byte of i, thus $e(i) = b(i) + s(i)$.

DEFINITION 1 (SEEK DISTANCE). *Given two data objects i and j, where i is the data object that has just been read, and j is the data object to be read next. The seek distance from data object i to j is defined as $dist(i, j) = |b(j) - e(i)|$.*

DEFINITION 2 (SEEK COST). *Given two data objects i and j, where i is the data object that has just been read, and j is the data object to be read next. The seek cost from i to j is modelled as*

$$Cost(i, j) = f(dist(i, j)). \tag{1}$$

A direct consequence of our approximation is that function f is not linear and may vary across different disk models from various manufacturers (see Appendix A). In the paper, this issue is tackled by an empirical approach: on a specific HDFS system using a certain category of disks, we run a large number of experiments measuring disk seek response time and seek distance to fit the non-linear function f. Our layout optimization algorithms, on the other hand, are agnostic to a specific form of f. They can be used in any HDFS-based column store, as long as its function f is measured. Details on how to model function f will be discussed in Section 5.

2.2 Table Layout in a HDFS Column Store

Column stores are the de-facto design of HDFS-based data analytical systems. Representative systems include RCFile [29], ORC File [5] and Parquet [3]. In a HDFS-based column store, a table is horizontally partitioned into row groups, each containing a number of rows. Within each row group, data is organized by columns. Each column is compressed and stored sequentially. These columns are indexed by a small piece of meta-data at the end of the row group, so that each column can be located efficiently. At the physical level, one row group generally fits into one HDFS block, the physical unit of HDFS data. A HDFS block is stored as a local file in the disk drive of a data node. In practice, HDFS-based column stores often recommend that a HDFS block holds only one row group [3, 30]. For sake of simplicity, we follow this convention in the following and use a row group and a HDFS block interchangeably. A high-level picture of the table layout in a HDFS-based column store is given in Figure 1.

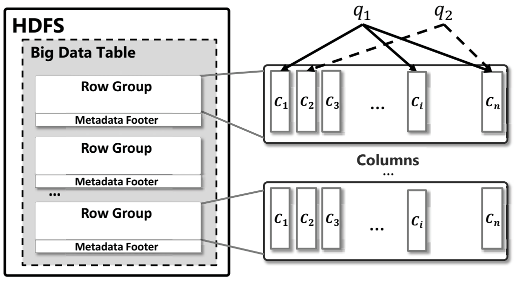


Figure 1: An illustration of the table layout in a HDFS-based column store, where c_i is a column and q_i is a query.

In today's column stores, columns in a row group are ordered as defined in the table schema. We refer to this order the *default* order.

We aim to find a new order algorithmically, such that total seek cost of all queries can be reduced and therefore improves the overall I/O performance. Column order is formally defined as follows.

DEFINITION 3 (COLUMN ORDER). *Given a table with n columns, $1, 2, \ldots, n$, a column order $S = \{c_1, c_2, \ldots, c_n\}$ is a permutation of the column set where $1 \leq c_i \leq n$ and $c_i \neq c_j$ when $i \neq j$.*

A column order $S = \{c_1, c_2, \ldots, c_n\}$ dictates how columns are arranged in a row group. In most columnar layouts on HDFS, when data is read from the physical store, the data engine by default searches the needed columns in a row group from the first column to the last, a behavior that is not affected by order-sensitive operators, such as `ORDER BY` and `GROUP BY`. Indeed, in MapReduce-based analytical systems (e.g Spark, Hive), column reading and tuple reconstruction are done at the beginning of the map stage, while `ORDER BY` and `GROUP BY` are done in the shuffle and reduce stages where intermediate I/O and network communications are performed. The table scan and data shuffling are decoupled and the column reading order is not affected by the shuffle relied operators such as `ORDER BY` and `GROUP BY`. Therefore, we focus on the data layout optimizations without stepping into the optimization of query plan and execution.

3. COLUMN ORDER OPTIMIZATION

In this section, we first give a formal definition of the column ordering problem and show that computing an optimal order for a given query workload is NP-Hard. We then design a heuristic algorithm that achieves approximations of the optimal solution with reasonable efficiency for production workloads.

3.1 Column Ordering Problem

The goal of column ordering is to improve I/O efficiency by optimizing the table layout according to the data access patterns of a query workload. In the workload of wide table analytics, a query q typically accesses only a small subset of all columns. Let $C_q = \{c_{q,1}, c_{q,2}, \ldots, c_{q,m}\}, m \leq n$ be the sequence of columns that q accesses (i.e., a column access pattern) following the column order S. Notice that our approach can capture arbitrary column access pattern.

Let $c_{q,i}$ be the i_{th} column accessed by query q. Given two columns $c_{q,i}, c_{q,j}$ accessed by query q, $c_{q,i}$ must appear before $c_{q,j}$ in S if $i < j$. In other words, the read task of query q fetches the required columns following the column order S, and will not yield zig-zag patterns during the table scan. Such column access pattern has minimal seek cost when reading columns in a row group, and it is applied in column stores such as Parquet [3]. A seek will occur if columns $c_{q,i}$ and $c_{q,i+1}$ are not adjacent in S. As such, the seek cost of a query is defined as:

DEFINITION 4 (QUERY SEEK COST). *Given a column order $S = \{c_1, c_2, \ldots, c_n\}$, and the column access pattern $C_q = \{c_{q,1}, c_{q,2}, \ldots, c_{q,m}\}$ of query q, over a table with N row groups, the seek cost of q is*

$$Cost(q, S) = N \times (\epsilon + \sum_{i=1}^{m-1} Cost(c_{q,i}, c_{q,i+1})), \tag{2}$$

where ϵ is the initial seek cost of seeking to the first column in a row group.

The initial seek cost ϵ is incurred when seeking from one row group to another and from the metadata footer to the first column in a row group. In some data analytical systems like Spark, the metadata footer can be cached in memory so that the seek cost from the metadata footer to the first column is eliminated. During query execution, the read sequence of the row groups is arbitrary so that the initial seek distance is random. Initial seek cost therefore can be approximated as the average seek cost of the disk. Note that every row group in a table has the same size and columns in each row group also have very similar size. That is why we can calculate the query seek cost by multiplying the seek cost of one row group by N.

Given a workload Q containing a set of queries and a column order S, the seek cost of the workload is defined as:

DEFINITION 5 (WORKLOAD SEEK COST). *Given a weight w_q for each query $q \in Q$, the seek cost of the workload Q is*

$$Cost(Q, S) = \sum_{q \in Q} (w_q \times Cost(q, S)). \tag{3}$$

The weight w_q captures the execution frequency or importance of query q.

When the number of columns of a table is small (e.g < 50), the seek cost may not be significant, compared to the sequential reading cost. However, in many real-world production environments, (e.g. the table of log analytics in Bing Microsoft), the number of columns exceeds 1000, and keeps increasing every month. In such scenarios, the seek cost becomes a major part of the I/O cost and indeed constitutes a significant fraction to the job's overall end-to-end latency. Finding an efficient column ordering therefore becomes essential. The problem is formally defined as follows:

DEFINITION 6 (COLUMN ORDERING PROBLEM). *Given a workload Q, find an optimal column order S^*, such that the seek cost of Q is minimized*

$$S^* = \arg\min_{S} Cost(Q, S). \tag{4}$$

THEOREM 1. *The Column Ordering Problem is NP-Hard.*

We prove Theorem 1 by reducing the classic Hamilton Path Problem [7] to the decision version of the column ordering problem. See proof in Appendix B.

3.2 Column Ordering Algorithm

We propose a probabilistic algorithm called SCOA (Simulated Annealing based column ordering algorithm) to solve the column ordering problem. As shown in Algorithm 1, SCOA leverages basic properties of Simulated Annealing (SA) [35] to solve the column ordering problem and generates an approximation of the optimal column order.

In SCOA, a specific column order corresponds to a state of the algorithm. Intuitively, if there is a 'good' column order S, it is likely that there is a better state S' 'close' to S (S' and S are similar in terms of the orders of columns). Such a heuristic greatly reduces the search space without searching all possible states. In SCOA, the energy of a state is measured by its seek cost. Different from other simple heuristic algorithms, SCOA makes decisions between accepting or rejecting the neighbor state probabilistically. The distribution of the acceptance probability is determined by the annealing schedule. This ensures that SCOA is unlikely to be trapped into an undesirable local minimum.

Given a workload Q, and an initial column order S_0, SCOA returns an preferable state S, so that the total I/O cost of Q is significantly reduced. In Algorithm 1, the main loop (lines 2-8) shows the iterative search process based on an annealing schedule

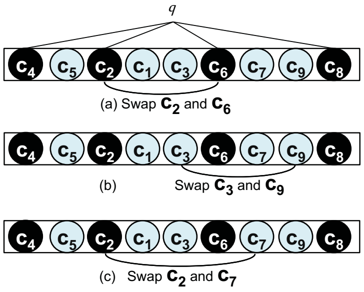
Algorithm 1: SCOA

```
Input: The set of queries $Q = \{q_1, q_2, ..., q_m\}$;
The initial column order $S_0 = \{c_1, c_2, ..., c_n\}$
Output: The optimized column order $S$;
$S := S_0, e := Cost(Q, S_0), t := t_0$;
for $k := 1$ to $k_{max}$ do
    $t := Temperature(t, cooling\_rate)$;
    $S' := Neighbor(S)$;
    $e' := Cost(Q, S')$;
    if $(e' < e) || (exp((e - e')/t) > random(0, 1))$ then
        $S := S'$;
        $e := e'$;
return $S$;
```

proposed in [35]. The *Temperature* function is the core function of the annealing schedule. In this algorithm, the temperature shrinks at a rate of $(1 - cooling\_rate)$. Function *Neighbor(S)* is to generate a candidate neighboring state from the current state S, achieved by swapping the positions of two randomly picked columns in S.[1] Parameter settings of SCOA are discussed in Appendix C.

3.3 Incremental Computation of Seek Cost

When the access pattern of a query follows the global column order (as adopted by existing systems such as HDFS), we can incrementally compute the seek cost of a query to speed up SCOA, given that a neighboring state S' is derived from the current state S by randomly swapping two columns. Consider the example in Figure 2. Query q accesses 4 columns $C_q = \{c_4, c_2, c_6, c_8\}$. When deriving a new state by swapping two columns in C_q (e.g., c_2 and c_6 in Figure 2(a)), the seek cost of this query clearly remains unchanged (both equal to $f(s(c_5)) + f(s(c_1) + s(c_3)) + f(s(c_7) + s(c_9))$, for reading a row group).



(a) Swap c_2 and c_6

(b) Swap c_3 and c_9

(c) Swap c_2 and c_7

Figure 2: Three cases of the delta query cost

A more complex case occurs when neither of the two swapped columns is accessed by the query q (e.g., c_3 and c_9 in Figure 2(b)). The pseudo code for handling this case is presented in Algorithm 2. The $SeekCost2ndCase$ function takes as input the current state S and two swapped columns c_x and c_y, and outputs the seek cost of the neighboring state S' for q. Let $suc(c_i)$ be the first succeeding column of c_i in C_q, and $pre(c_i)$ be first preceding column of c_i in C_q. For example, in Figure 2, $suc(c_1) = c_6$ and $pre(c_1) = c_2$. According to Algorithm 2, it is clear that $Cost(q, S') = Cost(q, S)$ if $suc(c_x) = suc(c_y)$. Otherwise, at most two terms in Equation 2 will be affected and it will be updated according to Lines 4-6 and Lines 7-9, respectively.

Algorithm 2: SeekCost2ndCase

```
   Input: A query $q$ and sorted set $C_q$;
   Current column order $S$, and its seek cost $Cost(q, S)$;
   Two swapped columns, $c_x \notin C_q$ and $c_y \notin C_q$.
   Output: The seek cost of the neighboring state $S'$,
           $Cost(q, S')$
1  if $suc(c_x) = suc(c_y)$ then
2      return $Cost(q, S)$;
3  $delta := 0$;
4  if $pre(c_x) \neq null$ and $suc(c_x) \neq null$ then
5      $delta -= f(b(suc(c_x)) - e(pre(c_x)))$;
6      $delta += f(b(suc(c_x)) - e(pre(c_x)) - s(c_x) + s(c_y))$;
7  if $pre(c_y) \neq null$ and $suc(c_y) \neq null$ then
8      $delta -= f(b(suc(c_y)) - e(pre(c_y)))$;
9      $delta += f(b(suc(c_y)) - e(pre(c_y)) - s(c_y) + s(c_x))$;
10 return $Cost(q, S) + delta$;
```

The last case occurs when exactly one swapped column is accessed by q (e.g. Figure 2(c)), which can be handled in a similar way to Algorithm 2. An important difference from the previous two cases is that C_q will be updated if the SA algorithm accepts this neighboring state S'.

Time Complexity. To maintain the sorted set C_q efficiently, we use a binary balanced search tree to insert, remove and query preceding and succeeding elements. All these operations run in $O(\log R)$ time. The overall time complexity of computing seek costs is $O(|Q| \cdot \log R)$, where R is the average number of columns accessed by a query. Compared to the naive approach of sorting all the columns for every new ordering, this incremental approach is R times faster. On the production data we tested, R is 32 and SCOA with incremental seek cost computation only requires a few minutes to converge.

Besides simulated annealing (SA), we have tried several other meta heuristics. Particularly, we have also tried to apply genetic algorithm (GA) [37, 51] in Appendix D and AutoPart [41] algorithm in Appendix E. Results show that SA performs much better.

4. STORAGE CONSTRAINED COLUMN DUPLICATION

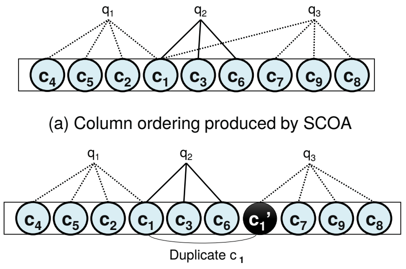
Suppose we have extra storage headroom, we may be able to further reduce the overall seek cost by duplicating some popular columns and inserting them into carefully selected positions within the derived column orders. Consider the simple example in Figure 3. In Fig. 3(a), the seek cost of both q_1 and q_2 is 0 while the seek cost of q_3 is $f(s(c_3) + s(c_6))$ (Note that the initial seek cost ϵ can be ignored as it is constant). In Fig. 3(b), however, if we duplicate c_1, insert it between c_6 and c_7, and let q_3 access the new replica of c_1, the seek cost of all three queries becomes 0.

We formally define the column duplication problem as follows.

DEFINITION 7 (COLUMN DUPLICATION PROBLEM). *Given a workload Q and the storage headroom H, identify a set of duplicated columns with an ordering strategy S_D such that 1) the total size of duplicated columns is not greater than H and 2) the seek cost of Q is minimized.*

In this section, we first introduce the basic idea of the duplication process in Section 4.1 and then provide details of how to optimize it in Section 4.2.

[1] We have also tested various other neighboring state selection heuristics, including substantially more complicated ones. However, none of them outperformed the simple 'column-swap' heuristic. For the sake of simplicity, we thus limit ourselves to the presentation of this most basic version of the algorithm.



(a) Column ordering produced by SCOA

(b) Duplicate c 1 and insert it between c 6 and c 7

Figure 3: An example of column duplication

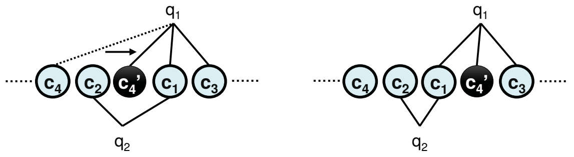
Algorithm 3: Duplication Algorithm

```
   Input: Column order $S_O$ produced by SCOA;
   Additional storage headroom $H$;
   Workload $Q$.
   Output: A column order $S_D$ containing duplicated columns.
 1 Calculate $C_q$ for each $q$ based on $S_O$;
 2 $S_D := S_O$
 3 $used\_volume := 0$;
 4 while $used\_volume < H$ do
 5     for column $c$ in $S_O$ do
 6         for $pos$ := 0 to $|S_D|$ do
 7             $cost\_reduced = cost\_reduce(S_D, c, pos, Q)$;
 8             if $cost\_reduced > max\_reduce_c$ then
 9                 $max\_reduce_c := cost\_reduced$;
10                 $best\_pos_c$ := pos;
11     $c\_dup$ := the column in $S_O$ with the largest $\frac{max\_reduce_c}{s(c)}$;
12     Insert $c\_dup$ into $S_D$ as the $best\_pos_{c\_dup}$ -th element;
13     $used\_volume += s(c\_dup)$;
14     Update $C_q$ for all $q$ in $Q$;
15 return $S_D$;
```

4.1 Duplication Algorithm

The basic idea of the duplication algorithm is to find and insert duplicated columns into the column order S_O produced by SCOA in a greedy fashion (see Algorithm 3). We repeatedly find a column to duplicate until the storage headroom is used up (Line 4). In each iteration, we enumerate the column to be duplicated (Line 5) and the position to which it will be inserted (Line 6). For each (c, pos) pair, we use the $cost\_reduce$ function to calculate how much absolute seek cost reduction is achieved when inserting a replica of column c as the pos-th element of the current ordering S_D. The variable max_reduce_c records the maximum seek cost reduction we can get when duplicating c and $best\_pos_c$ is the corresponding insertion place. We identify the column c_dup with the largest $\frac{max\_reduce_c}{s(c)}$ value among all columns where $s(c)$ is the column size, and insert it into S_D as the $best\_pos_{c\_dup}$-th element (Lines 11-12). After identifying c_dup and $best\_pos_{c\_dup}$, we need to update C_q if the seek cost decreases when q accesses the new replica c_dup (Line 14). The above algorithm without optimization is used as the ***baseline version algorithm***.

Note that we start from the column order S_O produced by SCOA because, intuitively, compared to other uninformative orderings such as the default column order, this optimized ordering enables us to select the most suitable (c, pos) pairs for duplication. An empirical evaluation also justified this intuition: under a constraint of 15% extra storage, the duplication algorithm starting from the default initial ordering achieved 10.6% average I/O performance gain while that starting from the optimized ordering achieved 12.4% average I/O performance gain (due to the randomness of the SA process, we ran the algorithms for 10 times).



(a) Duplicate c_4 to improve q_1 (b) Swap c_1 and c_4' to improve q_2

Figure 4: An example of periodical refine

4.2 Detailed Optimizations

4.2.1 Periodical Refinement

Duplicating a column c can help queries accessing c achieve smaller seek cost. On the other hand, queries not accessing c may experience larger seek cost if they have to seek over the new replica of c. We call this extra cost produced by duplication the *additional seek cost*. For example, in Figure 4(a), although duplicating c_4 reduces the seek cost of q_1 from $f(s(c_2))$ to 0, it introduces an additional seek cost for q_2 because q_2 has to seek over the new replica of c_4.

Figure 5 depicts the relationship between the relative seek cost reduction evaluated by the seek cost function and the number of duplicated columns of one run of our baseline approach (5% extra storage). We can see that at the beginning of the run, duplicating only a few columns achieves considerable seek cost reduction. This is because duplicated columns provide more options for queries accessing them and the additional seek cost incurred on queries not accessing them is negligible. However, as the number of duplicated columns increases, the additional seek cost incurred by duplicated columns becomes evident and counterbalances the benefit of duplication. Thus, the overall seek cost reduction due to duplication converges.

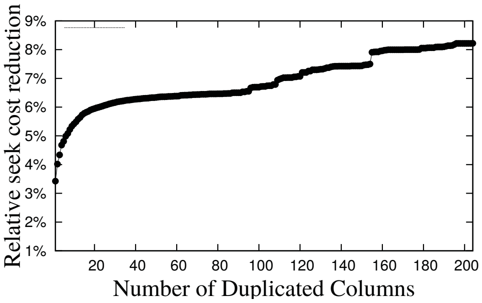


Figure 5: #Duplicated columns v.s. seek cost reduction

To decrease the additional seek cost and improve the achievable performance gain, we use the SA algorithm to periodically refine the ordering S' (e.g. each time after duplicating a fixed number of columns). The intuition is that SA refinement is able to adjust the relative positions of replicas with regard to the non-duplicated columns to reduce the additional seek cost. For example, in Figure 4(b), swapping columns c_1 and c_4' decreases the seek cost of both q_1 and q_2 to 0.

4.2.2 Running Time

Each iteration of the baseline algorithm runs in $O(n \cdot (n + d) \cdot |Q| \cdot R \log R)$, where n is the number of columns in the original table, d is the number of duplicated columns and R is the average number of columns accessed by a query. When n and d are large, this algorithm is not efficient.

Therefore, we provide two simple heuristics to tradeoff the duplication effect for running time efficiency. First, rather than enumerating all columns for possible duplication, we only consider the most popular ones. Here, we define the popularity of a column as the summation of the seek distances (Definition 1) from it to its neighbor columns (if two columns are both accessed by a query, they are neighbor columns). Second, instead of enumerating all possible insertion places, we can enumerate separated indices (e.g. indices that are a multiple of 10). The motivation is that inserting a replica into very close positions produces similar performance gain.

5. IMPLEMENTATION

In this section, we first present an overview of the adaptive data layout optimization solution that has been deployed in our real-world log analytical pipelines. Then we drill down to details of modeling the seek cost function $f(\cdot)$ and column redirection when column duplication is applied. An accurate seek cost function is the core of layout optimization while column redirection is necessary for query execution on duplicated layouts.

5.1 Adaptive Layout Optimization

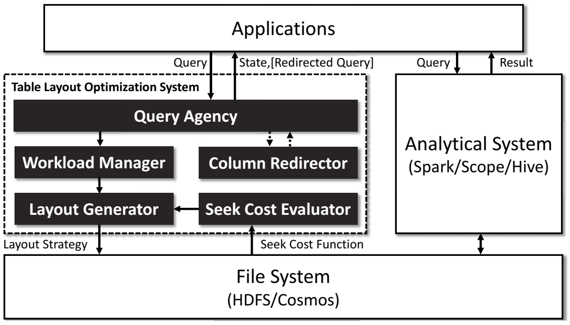


Figure 6: The framework of adaptive data layout optimization

To add data layout optimization feature to existing wide table analytical pipeline, there are some trade-offs to deal with in query performance, availability and system complexity. We successively follow the principles: 1) do not affect the availability of existing systems like Spark and HDFS so that our solution is safe to deploy, 2) minimize hacks to the source code of existing systems so that our solution is easy to deploy, and 3) minimize additional performance overhead produced by the new features. As shown in Figure 6, the adaptive data layout optimization solution has five key components:

Query Agency provides an HTTP interface to the applications. An application submits a query to *Query Agency* before it is submitted to the analytical system. Column duplication is enabled as a separate and optional feature in our solution. All wide tables have an ordered data layout without column duplication and may have an additional copy of data layout with duplication. *Query Agency* will check if column duplication can be applied to the query (whether a query can apply column duplication is discussed in Appendix F) and returns the result (state) in the response to application. When the state is $True$, which means that column duplication can be applied to the query, the *Query Agency* will get the redirected query (in which columns are redirected to the appropriate replicas) from *Column Redirector* and attach the redirected query in the response.

Workload Manager continuously maintains the most recent workloads Q within a certain period. Queries from applications are loaded into the *Workload Manager* by the *Query Agency*. Query workload statistics such as their frequency, the number of accessed columns and timestamps are maintained. These statistics are used to assign different weights to prioritize the queries. The workload statistics will be used by the *Layout Generator* to periodically produce a new data layout. The access pattern of each query is also maintained by the Workload Manager.

Column Redirector is called by the *Query Agency* and is responsible for picking an appropriate replica for each accessed duplicated column. It returns a new query which accesses the appropriate column replicas. To minimize the additional performance overhead, in-memory data structures and efficient column replica lookup algorithm are designed to guarantee a low latency column redirection. Details are discussed in Section 5.3.

Seek Cost Evaluator approximates the seek cost function $f(\cdot)$ according to the underlying storage hardware. Details are discussed in Section 5.2.

Layout Generator implements the SCOA algorithm and the column duplication algorithm to generate a new data layout. It records the averaged seek cost C of the workload when a new data layout strategy is applied, and continuously monitors the average seek cost C' for queries in Q based on the current data layout and workload. Once the layout generator detects that $\frac{C'}{C}$ exceeds a certain threshold (e.g., 1.05), it assumes that the current data layout is not good enough for the current workload, leading to an adaptive re-ordering process for generating a new data layout. Details of how to update data layout without reloading the historical data are given in Appendix F.

To ensure the availability and simplicity, we do not use a query interceptor to make query rewriting transparent to users. In modern analytical systems like Spark and Impala, any slave node can be the driver of a query. Query result is directly delivered to applications through the driver. An interceptor can help user applications submit the query and make everything transparent. But it may become the only single point of failure and the performance bottleneck. In case that query agency in the optimization system fails, applications can short-circuit the optimization system and submit queries directly to the analytical system and get benefits from ordered (without column duplication) layout (discussed in Appendix F). Our solution is easy to deploy and works with existing systems.

5.2 Seek Cost Evaluation

The seek cost evaluator is responsible for deriving the seek cost function (Definition 2), according to the underlying hardware and file systems. As discussed in Appendix A, for a given hard disk, the seek cost can be modeled as a function of seek distance [22, 42, 31]. But deriving an effective seek cost function is challenging, due to that seek cost is also affected by the relative position of cylinders [22] as well as many specifications of disk and system, which are hidden to applications. We therefore designed an empirical solution to evaluate the seek cost statistically. In the solution, a number of real seek operations at the same specific distance are performed on HDFS files. The average seek time of each specific distance d is used as the statistical seek cost of d. Linear interpolation is then applied to derive a curve of the seek cost function from the statistical seek costs. We then obtain a piecewise seek cost function from the real disk without knowing the hidden disk parameters and system characters. Figure 7 demonstrates the seek cost function obtained over three typical hard disk models listed in Table 1.

Table 1: Parameters of three disk models

Model	Disk Interface	Capacity (GB)	Spindle Speed (rpm)
SAS-2T (ST2000NM0033[13])	SAS	2000	7200
SAS-900G (ST9900805SS[12])	SAS	900	10000
SATA-2T (ST2000DM001[11])	SATA	2000	7200

The basic form of seek cost function has been discussed in the hardware community [31, 42, 22]. An example form of theoretical function is shown in Figure 7 to verify that the seek cost function derived from our solution is reasonable. Details about the theoretical seek cost function are discussed in Appendix A.

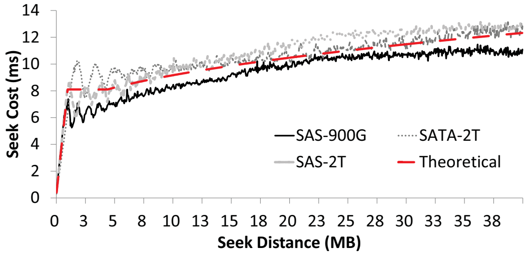


Figure 7: Seek costs of three different disks v.s. that of a theoretical model

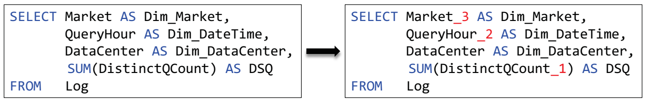
When evaluating seek cost on HDFS, we only perform in-block seek operations which is most commonly used in real data analytics. In Figure 7, we use the average seek cost of 100 seeks of the specific distance starting from different offsets inside different HDFS blocks. The difference between two adjacent seek distance is 50KB. We can see that the seek cost of all disks trend to increase as the seek distance increases. But in the first several megabytes of seek distance, this monotonic pattern is not always true. This is due to the effects of platter rotation. As discussed in Appendix A, when the seek distance is relatively small, the seek operation is likely to seek only a few cylinders so that the delay of arm movement is dominated by a relatively small settle time. The delay of platter rotation become a major part of the seek cost. Although the initial offset of the seek is random in our solution, with the same seek distance, there is a statistical expectation of the rotation delay. Especially when the seek distance is smaller than the capacity of a cylinder (1-2MB for the three disk models here), the seek operation is likely to be fitted in the same cylinder so that the seek cost become linear to seek distance which is also linear to the angle of platter rotation. Such a feature is important for column layout optimization because in a row group (eg. 128MB) with more than 1000 columns and a minority of the columns being large string columns, most columns are much smaller than 1-2MB. This gives us more opportunities to optimize the column ordering by putting frequently accessed columns nearby. Other disk parameters such as spindle speed and the number of sectors per track can affect the shape of the function curve. For example, disks with higher spindle speed usually have lower maximum seek time (cost), so that the seek cost function curve may converge to a lower maximum seek cost (Figure 7).

5.3 Column Redirection

When column duplication is applied, some columns may have more than one replicas. When a query arrives, the system rewrites the column names to their proper replicas. The replica selection strategy will affect the query performance. Beside analytical queries, data loading statements (in the form *INSERT INTO TABLE... SELECT... FROM...*) are also rewritten to duplicate the columns. This is simply done by replacing the original column name in *SELECT* with *column_name AS replica_name*. So we mainly discuss column redirection for analytical queries.

Given that the duplication algorithm has already output the optimal <column set, access pattern> hash map PL, where column set is the set of columns (not column replicas) to be accessed by a query and access pattern is the sequence of columns or column replicas to be accessed by a query. For the query whose column set exists in the current workload, we can find its access pattern directly from PL. For the query with a new column set, it is costly to find an optimal access pattern by enumerating all possible access patterns. Therefore, we need an efficient mechanism to find an viable replica access pattern.

Our solution is to maintain an inverted bitmap index in memory by *Column Redirector*. The bitmap index has a bitmap for each column (not column replica). The bitmap has a bit for each query to indicate whether or not the column is accessed by the query. The bitmap index is rebuilt periodically with the updates of table layout. When serving a query q without exact match in PL, *Column Redirector* looks up the index and select the bitmaps of the columns that are accessed by q. These selected bitmaps are merged by bitwise AND operation to find the query whose column set is a superset of q's. If more than one such query is found, the query with smallest column set is chosen. Then the access pattern of the query is used to generate the access pattern of q. If no such query can be found, we search for the query with most number of common columns as q. This is done by counting the number of set bits (TUREs) on each specific bit in the selected bitmaps. It is more efficient than calculating union of the queries' column sets. It can be done within milliseconds. The access pattern of the retrieved query is used to generate the access pattern of q. In this case, there will be some unmatched columns within q. We perform a first-fit strategy which scans the column order sequentially and get the first occurred column/replica as the right column/replica of each unmatched column. After the access pattern is determined, the original column names in q are replaced with the names of corresponding column replicas. As shown in Figure 8, if the 3rd, 2nd and 1st replicas of column *Market*, *QueryHour* and *DistinctQCount* are the best ones for this example query, the redirection result can be found in the right part of the figure. Then the rewritten query is returned to the application through *Query Agency*. While *Column Redirector* will continue searching the best access pattern for q , add it to PL and update the bitmap index, so that q and other new queries may directly benefit from it next time.



```
SELECT Market AS Dim_Market,
       QueryHour AS Dim_DateTime,
       DataCenter AS Dim_DataCenter,
       SUM(DistinctQCount) AS DSQ
FROM   Log
```

```
SELECT Market_3 AS Dim_Market,
       QueryHour_2 AS Dim_DateTime,
       DataCenter AS Dim_DataCenter,
       SUM(DistinctQCount_1) AS DSQ
FROM   Log
```

Query Re-write by column Re-direction

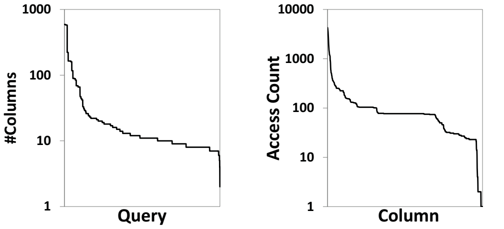
Figure 8: Query re-write example

6. PERFORMANCE EVALUATION

In this section, we evaluate the effectiveness and efficiency of our column ordering and duplication solutions. All experiments are done on real-life production workloads and log data.

6.1 Production Data and Query Workloads

The real-life production workloads and data we use are from Bing Microsoft's Web search data mining pipeline. Some statistics of data (one specific table) and workloads collected by our *Workload Manager* are listed in Table 2. Most of the queries in the workloads are aggregation queries with templates like *SELECT... FROM... [WHERE... GROUP BY... ORDER BY...]*.



(a) Number of columns accessed by query (b) Column's access count

Figure 9: Statistical patterns of the workload

Considering both query workload and the data layout, we summarize the statistical workload patterns in Figure 9. As shown in Figure 9(a), the number of columns accessed by a query follows a long-tail

Table 2: Statistics of Production Data and Workloads

Feature	Statistics
Column Count	1187
Column (Data Type, count)	(int,1008)(string,109)(boolean,36) (double,23)(long,9)(datetime,2)
Query Count	4000+ queries per month
Data Size	about 5T new data per day
Schema Update Frequency	schema slightly changes within 3-6 months
Query Update Frequency	less than 5% queries change within a month

distribution. The ‘smallest’ query accesses only 2 columns while the ‘largest’ query accesses 591 columns which is half of the whole table. This is one of the reasons of not vertically partitioning the table - maintaining a wide table can reduce the join cost compared to vertically partitioning the table into narrow ones. 77.5% of queries access 7 to 20 columns. The average number of columns accessed by a query is 32. Figure 9(b) summarizes the number of queries that each column is accessed by, which also follows a long-tail distribution. We analyze 4343 queries subscribed within a month. The most popular column is accessed by 4267 queries. There are 10 columns accessed by more than 1000 queries and 60% of the columns are accessed by 50-150 queries. On average, each column is accessed by 128 queries.

As shown in the above statistics, wide table analytical workloads differ substantially from analytical workloads (e.g., TPC-H [14] and SSB [10]) in traditional data warehouse. In TPC-H, there are 22 queries, the fact (largest) table has only 16 columns. In SSB, there are 13 queries, the fact table has only 17 columns. Therefore, existing solutions which work well on traditional data analysis may not be effective when being applied to wide tables.

6.2 Evaluation Settings

The machine used in the evaluation is equipped with two Intel Xeon E5-2670 2.6GHz CPUs (16 cores and 32 threads in total) and 24GB memory. One SAS-900G disk and one SAS-2T disk listed in Table 1 are directly attached for data storage without RAID configured as RAID is not recommended for HDFS [50]. SAS-2T is used as the default disk in our evaluations. The OS of the machine is CentOS Server 6.4 and the kernel release is 2.6.32-358.el6.x86_64. The disks are formatted with ext4 file system. The versions of the systems used in the evaluations are: Hadoop (1.2.1), Parquet (parquet-hadoop-1.6.0.rc4) and Spark (1.2.0). Three types of environments are set-up for our evaluations:

- **Single Node α:** single node HDFS + Spark. The data replication factor of HDFS is 1;
- **Cluster β:** 5-node HDFS + Spark (one node is assigned as the HDFS namenode and Spark master, the other four nodes as HDFS datanodes and Spark slaves). The network is Gigabit Ethernet. The data replication factor of HDFS is 3;
- **Cluster γ:** 64-node HDFS + Spark (one node is assigned as the HDFS namenode and Spark master, the other nodes as HDFS datanodes and Spark slaves). The network is Gigabit Ethernet. The data replication factor of HDFS is 3. We change the number of slave nodes (HDFS data nodes and Spark slaves) in this cluster to perform the evaluation at different scales.

Single Node α and Cluster β use separate machines which are not in the production cluster. Cluster γ shares machines with production cluster. To ensure that our experiments do not interfere with production workloads, we perform most of the experiments on Single Node α and Cluster β. Experiments on Cluster γ are performed when the production cluster is idle.

In this section, four column ordering strategies are evaluated:

- **Default:** the default column order used in production;
- **Naive:** frequently used columns are in the front of row group;
- **AutoPart-C:** the column ordering algorithm based on AutoPart [41]. See details in Appendix E;
- **SCOA:** our ordering algorithm proposed in Section 3.

In the evaluations, queries with the same column access pattern are considered as the same type of queries and only the one with the highest weight is evaluated. So that only 547 queries out of 4343 are evaluated. Late materialization [18] of the query results is enabled as the default feature in wide tables stored as Parquet format.[2]

6.3 Effects of Column Ordering Solutions

In this experiment, we evaluate the I/O efficiency of different column ordering solutions. The experiments are conducted in Single Node α using 1.2TB (390GB after encoding) data. We compare SCOA with the *Default*, *Naive* and *AutoPart-C*. The read latency is measured from the latency of Parquet Reader which directly reads the parquet files from HDFS. According to the results of Figure 10, if we take *Default* as the baseline, on average, *Naive* reduces 25.3% read latency, *AutoPart-C* reduces 43.7% read latency and *SCOA* reduces 65% read latency. If we take *Naive* as baseline, on average, *AutoPart-C* saves 23.4% read latency while *SCOA* reduces 52.7% read latency. If we take *AutoPart-C* as the baseline, *SCOA* reduces 38.1% read latency. Compare to *Default*, the maximum gain of *SCOA* is 73.9% and the minimum gain is 2.5%, which means that every single query benefits from the re-ordering. Compare to *AutoPart-C*, the maximum gain of *SCOA* is 57.9% and the minimum gain is -14.1%. *SCOA* is only outperformed by *AutoPart-C* on three queries. Actually, we have not ever encountered a case that default column order was better the *SCOA* optimized one. Since the performance of *Default* is very poor, in the following evaluations, we take the column order from *Naive* as the baseline for evaluation.

The experiments in Figure 10 are tested in Single Node α with a single thread Parquet Reader. To confirm the effectiveness of *SCOA* in real query execution, we conduct the experiments in the same environment with real Spark SQL queries (Figure 11). We configure 64 threads (tasks) per node in Spark. Elapsed time of table scan (map) stage of the queries is recorded as query read latency. If we take Naive as the baseline, *SCOA* reduces 42.0% read latency on average while *AutoPart-C* reduces 17.9%. If we take *AutoPart-C* as the baseline, *SCOA* outperforms it by 28.3% on average. The gain in Spark is lower than that of a HDFS file reader. This is caused by additional overheads beyond I/O, which is discussed in Section 6.4.

Besides the performance gain, running time is another feature of the column ordering algorithms. We implemented both *SCOA* and *AutoPart-C* with Java. Taking the workloads and table schema in this paper as input, *SCOA* converges in a few minutes, while AutoPart-C takes about 3 hours with the same runtime settings.

6.4 Effects of Concurrent Reading Tasks

There are two main factors that affect I/O performance gain of column ordering on Spark: 1) I/O competition of concurrent reading tasks; and 2) Overhead beyond reading, such as task scheduling, map-side shuffle write and Java garbage collection. The concurrent reading tasks may compete for I/O channels and affect the read behaviors of the disk. The overhead beyond reading may reduce the proportion of I/O cost in the query's end-to-end latency thus reducing the overall gain of column ordering solutions.

To understand the impact of concurrent reading tasks, we conduct an evaluation with different number of concurrent tasks (threads) on Spark in Single Node α. 1.2TB (390GB after encoding) data is used. The result is shown in Figure 12. *Naive* is used as the baseline.

[2] In parquet, columns in a row group are read and filtered before reconstructing them into records [4, 3].

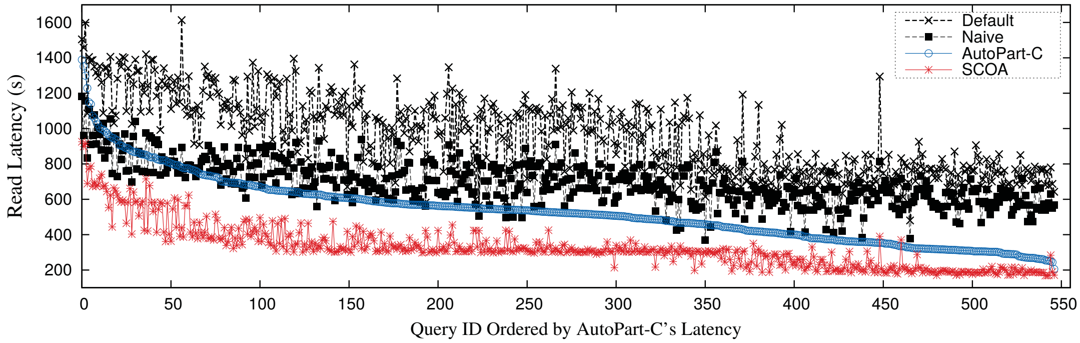


Figure 10: Read performance of different solutions on Single Node HDFS

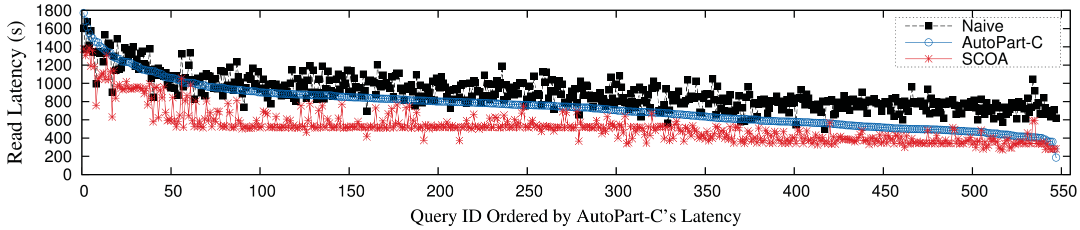
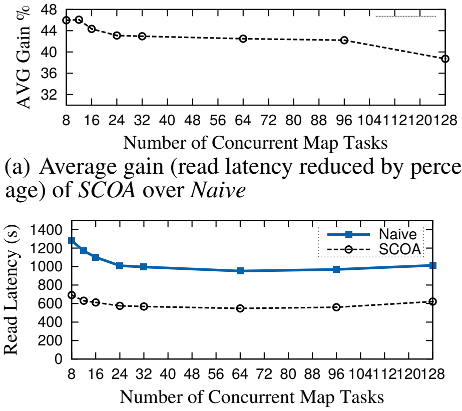


Figure 11: Read performance of different solutions on Single Node Spark

It can be seen in Figure 12(a) that the gain of *SCOA* remains stable (above 42%) as the number of concurrent reading tasks ranges from 8 to 96. The machine used in the evaluation has 16 cores and 32 physical threads. As shown in Figure 12(b), when increasing the number of concurrent tasks from 8 to 64, the query latency decreases, due to better parallelism. After that, with 96 and 128 concurrent tasks, the latency increases. This is due to the negative effect of I/O competition among concurrent threads. It confirms that I/O competition of concurrent reading tasks does have an effect, but it does not significantly affect the gain of column ordering when operated with a reasonable number of concurrent reading tasks.

The comparison of *SCOA*'s gain over *Naive* in Figures 10 (with 52.7% average gain) and 11 (with 42.0% average gain) shows that overheads beyond reading drastically affect performance. Figure 10 is evaluated by calling the API of Parquet without task scheduling, map-side shuffle write and Java garbage collection overheads while Figure 11 is evaluated on Spark with these overheads.



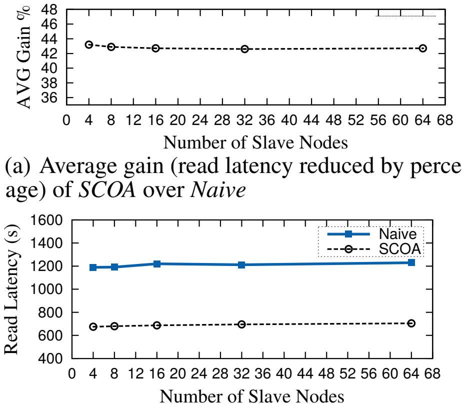
(a) Average gain (read latency reduced by percentage) of *SCOA* over *Naive*

(b) Average read latency

Figure 12: Column ordering's performance of different number of concurrent map (reading) tasks in Spark

6.5 Effects of Different Cluster Sizes

To evaluate the effects of different cluster sizes, we perform evaluations on Spark in production Cluster γ with 64 threads (tasks) per node. The data size is 1.2 TB per node. The result is shown in Figure 13. The x-axis is the number of slave nodes in the cluster and the y-axis is the average gain of *SCOA* compared to *Naive*. To reduce the evaluation cost and minimize interference with the production workload, we pick 50 most frequently issued queries from the whole workloads. It can be seen from Figure 13(a) that the gain is stable (around 43%) under different cluster scales. It can be seen from Figure 13(b) that the query's read performance does not degrade significantly when the cluster scales out. When the cluster scales from 4 nodes to 64 nodes, average query latency of *SCOA* increases by 4.31%, while average query latency of *Naive* increases by 3.35%. We can conclude from the evaluation that performance gain of column ordering is stable with different cluster sizes.



(a) Average gain (read latency reduced by percentage) of *SCOA* over *Naive*

(b) Average read latency

Figure 13: Column ordering's performance of different cluster scales in Spark

6.6 Effects of Different Row Group Sizes

For a table of certain size, larger row group size results in larger columns inside the row group but smaller number of row groups. The seek distance inside a row group and the total number of seeks differs for different row group sizes. In this experiment, we compare the read performance of both *SCOA* and *Naive* under three different row group sizes. Since there is only one row group per HDFS block and the row group size approximately equals to the HDFS block size (the metadata footer in each HDFS block is relatively very small), the effects of different block size can be addressed by this experiment. The evaluations are conducted in Single Node α with Parquet Reader. 1.2TB data is used in the evaluations. In Parquet, the row group size is controlled by the parameter *parquet.block.size*, which is the size of a row group object in memory. When writing data into Parquet format, the data is first written into memory and after an entire row group is constructed, it is then flushed into HDFS. Figure 14 demonstrates the performance of different row group sizes. For better visualization, we only report the results of the first 10 frequently executed queries.

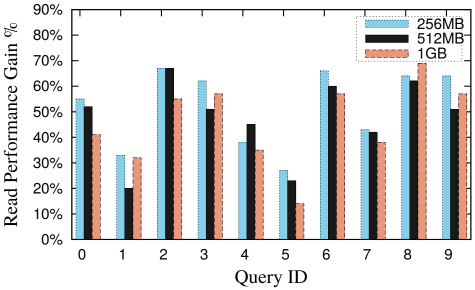


Figure 14: Read performance gain (read latency reduced by percentage) of *SCOA* over *Naive* with different row group sizes

From Figure 14, for different row group sizes, 256MB, 512MB and 1GB, *SCOA* shows significant performance gain over *Naive*. For the whole workload, average gains of *SCOA* under the three row group settings are 52.7%, 48.0% and 45.8% respectively, while average read latencies of *SCOA* are 344.3s, 252.4s and 196.7s respectively. As the row group size increases, the average gain decreases. This is because that to read a table of specific size, the larger row group size leads to less row groups and less number of total seeks (the number of seeks inside a row group is the same with the same column order) while the total seek distance is the same. As shown in Section 3.1, the seek cost function is not linear, two short seeks are likely to have higher cost than one long seek with the same total seek distance. That is why the average read latency of both *SCOA* and *Naive* under 512MB and 1GB row group settings are smaller than that of 256MB. The default value of *parquet.block.size* is 128MB and the maximum value to set is 2GB. In our production environment, we use a relatively larger value for wide tables. But larger row group size leads to much more memory consumption during data loading and query processing. Therefore, 256MB and 512MB are appropriate values for most cases. 256MB row group size is used by default in our experiments.

6.7 Effects of Different Seek Cost Functions

Seek cost function is a core part for our table layout optimization. In this experiment, we evaluate the effects of different seek cost functions. As discussed in Section 5.2, the seek cost functions for *SCOA* are obtained by performing seek and read operations on the real data in HDFS. Here we use an intuitive linear seek cost function in SCOA (*SCOA-Linear*) and compare it to *SCOA* with seek cost function derived by *Seek Cost Evaluator* (*SCOA-Evaluated*). Note that the linear seek function is not correct; we compare it to SCOA in order to highlight the importance of a correct cost function. The evaluation is conducted with Parquet reader and Spark SQL queries in Single Node α with 1.2TB data. The results of end-to-end Spark queries are shown in Figure 15.

For the read latency on the whole workload evaluated by Parquet Reader, taking *Naive* as the baseline, *SCOA-Linear* has 42.4% gain while *SCOA-Evaluated* has 52.7% gain. For the query latency on the whole workload evaluated in Spark, taking *Naive* as the baseline, *SCOA-Linear* has 31.4% gain while *SCOA-Evaluated* has 39% gain. This verifies that seek cost function derived by our *Seek Cost Evaluator* can more accurately demonstrate to the seek cost in real data read tasks. In Figure 15, *SCOA-Linear* sometimes outperforms *SCOA-Evaluated*. This is because the column ordering algorithm aims to minimize the I/O cost of the *whole* workloads, not individually. So a small minority of queries may not benefit from *SCOA-Evaluated*.

Table 3: Column Duplication: percentage I/O (read performance) gain under different SCOA refine periods.

Refine Period	1	5	25	45	65	85
Average I/O gain	10.7%	10.0%	8.4%	6.2%	5.9%	4.6%

6.8 Effects of Different Disks

In this subsection, we show that *SCOA* works well on different disk models. We conduct a group of evaluations on the workloads with the same amount of data as shown in Figure 10 (Single Node α with 390GB encoded data), but the underlying disk for HDFS changes from SAS-2TB to SAS-900GB. Results show that *SCOA* on SAS-900GB disk also has significant performance gain. Taking *Naive* as the baseline, the average gain of *SCOA* on SAS-900G is 50.7%. On SAS-2TB disk, the corresponding gain is 52.7%.

6.9 Effects of OS Caching Policies

In this experiment, we compare the read latency of the queries with OS cache enabled and disabled. The evaluations are conducted in Single Node α with 1.2TB data. The queries are executed by Spark, and the elapsed time of the table scan stage is recorded as the read latency. The results are shown in Figure 16. It can be seen that comparing enabling to disabling OS cache, the results do not have significant difference. This is because the data size is much larger than the memory capacity and the OS cache is flushed. So we can conclude that the OS cache does not have significant impact on our solution in the production environment. Actually, the OS invests additional CPU and memory resources to maintain the cache. This can be an overhead in some large sequential reads over HDFS [43]. In Figure 16, some queries was indeed hurt by caching polices. But since most queries in our workloads are I/O intensive (not CPU intensive) the effect of OS caching is insignificant.

6.10 Column Duplication

Finally, we examine the effectiveness of our column duplication solution as introduced in Section 4. The duplication I/O gain is computed as 1-$latency_{duplication}/latency_{order}$ and measured from the average gain of the whole workloads. Experiments are conducted in Cluster β with 4.8 TB raw data. We first examine the effect of different refine periods (number of duplicated columns between adjacent SCOA refinements). Using the same ordering produced by the SCOA algorithm, we ran the duplication algorithm with refine periods 1, 5, 25, 45, 65 and 85 under a constraint of 15% extra storage to compare their I/O gains. Due to the randomness of the *SCOA* process, we ran the above process 5 times. The average I/O gain of each period is shown in Table 3. We can see from this table that the performance gain decreases as the period length increases, which shows the importance of timely SCOA refinement.

Figure 17 demonstrates the I/O gains over the column order generated from *SCOA* with different sizes of storage headroom. We compare the duplication algorithms with (refine period = 5) and without optimizations. As the storage headroom becomes larger, the gain keeps increasing. The gain with optimization is much larger than that of the baseline approach without optimization. From the result, we see that we can use less than 5% extra storage for an-

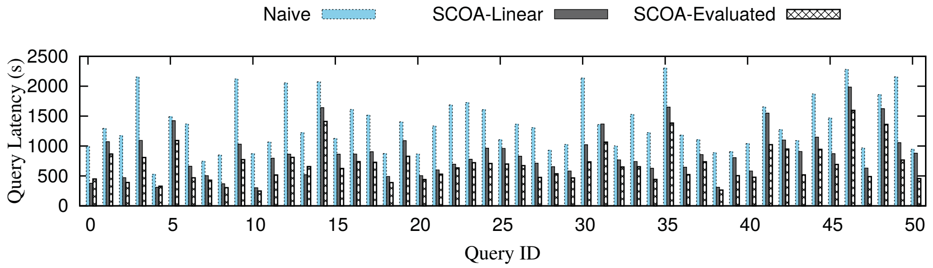


Figure 15: End-to-end query latency of SCOA with two different seek cost functions: SCOA-Linear for linear seek cost function and SCOA-Evaluated for seek cost function evaluated in read environment

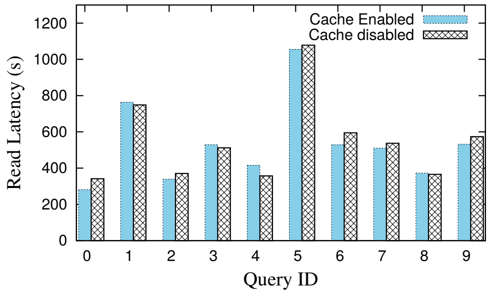


Figure 16: Read latency of different OS cache policies

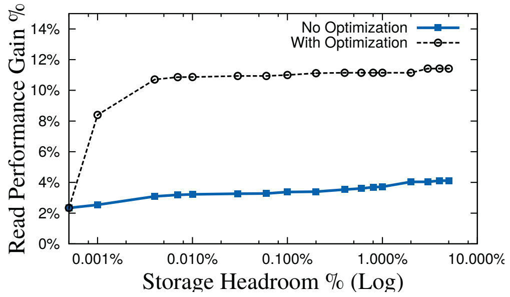


Figure 17: Read performance gain of column duplication with and without optimization

other 12% gain from duplication after column re-ordering. This confirms the effectiveness of our duplication algorithm. Actually, we also considered a column duplication algorithm based on AutoPart (discussed in Appendix E), but it does not provide any gain over AutoPart-C.

Except the read performance gain, there are some side effects of column duplication: 1) except the storage overhead, the data loading time is also increased; 2) the queries have to be redirected to read the right column replicas. The solution of redirecting column for queries is shown in Section 5.3. Our solution can rewrite the column names for query in milliseconds so that the overhead of column redirection is insignificant compared to the hundreds-of-seconds query execution time. We mainly evaluate the overhead in data loading. The results are shown in Table 4. In the evaluation, we load 4.8 TB raw data into the column-duplication table (in Parquet format) in Cluster β by Hive. 8 writing threads (map tasks) on each node is used for data loading. In Table 4, we can see for the table without duplication, data can be loaded in 296 minutes, when the columns are duplicated, the data loading time increase approximate-linear with the number of total columns in the tables. This is because data loading is main bounded by CPU time cost on encoding and packing data into columns.

7. RELATED WORK

For enterprise-level big data analysis (e.g, 100+TB of log data is generated every day at Bing Microsoft), I/O cost typically plays a major role in the overall computational cost. There are many efforts towards reducing the I/O cost of large scale data analytics.

Table 4: Elapsed time of loading data into the duplicated table

Storage Headroom	0	0.1%	1%	5%
#Cloumns	1187	1211	1291	1354
Data Loading Time	296min	301min	319min	335min

Big Data Analytical Systems: Hive [48] and Pig-Latin [40] are early big data analytical systems running on HDFS. They benefit from the fault tolerance and scalability of Hadoop to support SQL-like big data analysis on top of a large cluster of commercial machines. SQL queries are translated into MapReduce jobs to process the data on HDFS. Recent systems such as Presto [9], Impala [2], Drill [1] and Spark SQL [8] replace the early version of Hive, by using optimized query engines. They try to avoid dumping large amounts of intermediate results to disks by optimizing the shuffle strategies and utilizing memory more extensively. But HDFS is still used as the common underlaying distributed file system because it has become the defacto standard of big data storage. HDFS offers an inexpensive, scalable, fault-tolerant and generic storage layer for big data analysis. In Bing Microsoft's log analysis applications, Spark [53] is also widely employed. It is a full-stack system supporting SQL-based analysis, stream processing, machine learning and graph computation. Spark also reads original data from HDFS, but it is much more efficient than MapReduce in both I/O and computation aspects. These efforts on big data analysis systems can be considered as complementary works to the table layout optimization solution proposed in this paper, since our solution does not need to change any component of those systems.

Columnar File Formats on HDFS: Columnar storage has been considered as a very I/O efficient data layout for read-only analytical jobs [45]. As such, file formats such as RCFile [29], ORC File [5] and Parquet [3] apply columnar storage techniques [17, 15, 45] to HDFS, where the data is split into blocks without exposing the structure and contents of the blocks.

By applying columnar file formats, data analytical jobs on top of HDFS can avoid to read unnecessary columns and further benefit from type-specific data compression. In RCFile, data within an HDFS block is first partitioned into row groups [29] whose size is approximately 4MB for each. Within each row group, data is arranged in a columnar manner. A metadata header is stored in each row group for indexing the columns. ORC File is an improvement over RCFile. It applies a larger row group size (256 MB by default) to improve the I/O performance [30]. ORC File uses a technique called *data decomposing*[5] to decomposes complex data types such as *Map* and *Struct* to multiple columns with primitive data types. It also supports type-specific encodings for each column [5]. ORC File is now mainly supported by Hive. Parquet is similar to ORC File. It also uses a relatively larger row group size (128MB by default), and supports data decomposition and type-specific encoding [3] as well. Parquet is now widely supported by systems such as Spark, Hive, Impala, Pig-Latin and Drill [6]. These columnar formats on HDFS are also used in our analytical pipelines at Bing Microsoft. Our optimization solution can be applied to all the above columnar file formats.

Except applying columnar storage inside HDFS blocks, CIF [24] format stores columns of a row group in separate HDFS blocks and co-locate the blocks on the same node. This method has significant performance improvement compared to RCFile in small row group (4MB) settings. But this solution needs to hack into existing systems and it is shown in experimental study [30] that when large row group size (eg. 265 MB) is applied in later formats like ORC File and Parquet, it is not necessary to store columns of row group into multiple physical blocks.

Column Grouping Table Layout: While columnar layout is confirmed to be efficient enough for most analytical workloads [30, 33], there are some applications where queries are likely to access different sets of columns. Some studies [27, 34] try to utilize such access patterns to group some columns together. In Trojan [34], a column grouping layout on HDFS is proposed. However, the algorithm scales exponentially in time with the number of columns. Given a wide table with thousands of columns, this algorithm cannot be directly applied. But the idea of grouping columns, which is also used in data partitioning works such as AutoPart [41], is used in designing our baseline. Moreover, the idea of putting column groups of a row group within in HDFS block (no store them into separate blocks) is used in our baseline solution.

Horizontal Partitioning Optimization: As data is horizontal partitioned into blocks in HDFS-like systems, there are research works [46, 54] focused on optimizing horizontal data partitioning. Sun's [46] proposed a fine-grained blocking technique to better pack tuples into blocks and enable queries to skip blocks. Zhou's [54] discussed how partitioning techniques are used in data shuffling and query execution. Since we focus on table layout inside blocks and do not modify existing systems, these works can be complementary works and directly applied on top of ours to further boost query performance.

Beyond HDFS environment, database community has done extensive works on data partitioning [41, 28, 26, 38, 23, 20, 21, 32, 33]. HYRISE [26], Ailamaki's [20] and DataMorphing [28] focus on cache performance optimization. They reorganize data in memory to reduce cache miss of processors. Although they have a different objective, their methods have been learned by disk-based analytical databases [44, 33] and data layout designing on HDFS [29, 34]. Hill-Climb algorithm in DataMorphing [28] is evaluated to be very efficient in boosting I/O performance of disk-based analytical workloads [33]. Although it scales exponentially in both time and space with the number of columns [26] and thus cannot be directly used in wide table layout optimization, we use the idea of merging two columns/column groups at a time in designing the baseline.

Navathe's [38] and Schism [23] focus on data partitioning methods for OLTP workloads. Navathe's [38] studied how to maximize local and in-memory transaction processing by vertical partitioning data into fragments. Its partitioning algorithm is also used in O2P [32] to support online data partitioning in analytical workloads. Online and adaptive partitioning is also studied in H2O [21]. We learn lessons from adaptive storage and applied the methodology in our adaptive layout optimization framework. Schism [23] uses a graph-based algorithm to horizontally partition tuples into groups and minimize the number of distributed transactions. In addition, techniques for approximate query evaluation through sampling also target to optimize I/O performance [52, 19] by not reading the whole data. They are only applicable to aggregation queries and only provide approximate results.

AutoPart [41] proposed a workload-based vertical partitioning algorithm for large scientific databases. The algorithm is also evaluated to be very efficient in boosting I/O performance of disk-based systems [33]. The application and addressed problem of AutoPart are not the same as that in this paper. So that AutoPart can not be directly applied in column ordering/duplication. We tailored AutoPart and integrated it with Hill-Climb in designing the baseline in this paper. Similar to vertical data partitioning, some columnar databases such as C-Store [45] enable users to group different columns into multiple overlapping projections or materialized views. Each projection is sorted on the same attribute so that a query can be solved using the most advantageous projection without joining columns with different sort keys. Data in a projection is still stored in columnar manner. So even when the best projection contains unnecessary columns, the query does not need to read them. But for wide table workloads such as that in Section 6.1, building a projection for each query will cost 121 times more storage space which is impractical for large production data, while solving queries with limit storage overhead comes back to our storage constrained column duplication problem. Furthermore, store column groups into separate distributed files in HDFS may cause a query to perform distributed joins to reassemble tuples which is far more expensive than I/Os inside blocks [36]. Even store column groups into separate local files and co-locate them on the same node like [24] need to hack into existing systems and is proved to have not performance gains when large row group size is applied [30]. However, the idea of storing data inside a column group in columnar manner is used in our baseline solution.

8. CONCLUSION

Column store is widely used for efficient data analytics. The *order* in which columns are physically stored has not received any attention because it has been commonly assumed that the number of columns in a big table is small, and thus the impact of different column orderings is irrelevant. When analyzing multiple materialized views. Disk seek cost can be saved if the needed columns of a query can be just fitted in Bing Microsoft's search log, however, we found a large number of very wide tables with thousands of columns in the data analysis pipeline. In such wide tables, the order of columns can greatly affect I/O performance, and thus we propose in this paper to optimize the column order according to the disk access patterns derived from the workloads. Investing a small amount of extra storage headroom, we show that we can combine column duplication with column ordering to achieve further I/O gains. We verify our algorithms using real-world production workloads on a real implementation. The results show that after applying the proposed ordering strategies, we can save up to 73% query processing end-to-end cost compared to the default columnar layout. With less than 5% storage headroom, we achieve another 12% gain using column duplication.

9. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive comments. This work is supported by the National Science Foundation of China under grant (No. 61472426 and 61432006), Science and Technology Planning Project of Guangdong under grant (No. 2015B010131015), the Ministry of Science and Technology of China, National Key Research and Development Program (Project Number: 2016YFB1000700), and the Fundamental Research Funds for the Central Universities, the Research Funds of Renmin University of China No. 14XNLQ06. Du Xiaoyong's work is supported by ECNU-RUC-InfoSys Joint Data Science Lab.

10. REFERENCES

[1] http://drill.apache.org/.

[2] http://impala.io/.

[3] http://parquet.apache.org/documentation/latest/.

[4] https://berlinbuzzwords.de/sites/berlinbuzzwords.de/files/media/documents/ted_dunning-what_and_why_and_how_apache_drill.pdf.

[5] https://cwiki.apache.org/confluence/display/hive /languagemanual+orc.

[6] https://cwiki.apache.org/confluence/display/hive/ parquet.

[7] https://en.wikipedia.org/wiki/hamiltonian_path_problem.

[8] http://spark.apache.org/sql/.

[9] https://prestodb.io/.

[10] http://www.odbms.org/2014/03/star-schema-benchmark/.

[11] http://www.seagate.com/staticfiles/docs/pdf/datasheet/disc/barracuda-ds1737-1-1111us.pdf.

[12] http://www.seagate.com/staticfiles/docs/pdf/datasheet/disc/savvio10k5-fips-data-sheet-ds1727-4-1201-us.pdf.

[13] http://www.seagate.com/www-content/product-content/constellation-fam/constellation-es/constellation-es-3/en-us/docs/constellation-es-3-data-sheet-ds1769-1-1210us.pdf.

[14] http://www.tpc.org/tpch/.

[15] D. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[16] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column-oriented database systems. *PVLDB*, 2(2):1664–1665, 2009.

[17] D. J. Abadi, S. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD*, pages 967–980, 2008.

[18] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented dbms. In *ICDE*, pages 466–475, 2007.

[19] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42, 2013.

[20] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, pages 169–180, 2001.

[21] I. Alagiannis, S. Idreos, and A. Ailamaki. H2o: a hands-free adaptive store. In *SIGMOD*, pages 1103–1114, 2014.

[22] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Hard disk drives. In *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.80 edition, May 2014.

[23] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1-2):48–57, 2010.

[24] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. Column-oriented storage techniques for mapreduce. *PVLDB*, 4(7):419–429, 2011.

[25] K. S. G., H. S. G., and M. B. Taghadosi. Importance of the initial conditions and the time schedule in the simulated annealing. In R. Chibante, editor, *Simulated Annealing, Theory with Applications*, chapter 12, pages 217–234. Sciyo, August 2010.

[26] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. Hyrise: a main memory hybrid storage engine. *PVLDB*, 4(2):105–116, 2010.

[27] S. Guo, J. Xiong, W. Wang, and R. Lee. Mastiff: A mapreduce-based system for time-based big data analytics. In *CLUSTER*, pages 72–80, 2012.

[28] R. A. Hankins and J. M. Patel. Data morphing: an adaptive, cache-conscious storage technique. In *VLDB*, pages 417–428, 2003.

[29] 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*, pages 1199–1208, 2011.

[30] 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*, 6(14):1750–1761, 2013.

[31] D. M. Jacobson and J. Wilkes. *Disk scheduling algorithms based on rotational position*. Citeseer, 1991.

[32] A. Jindal and J. Dittrich. Relax and let the database do the partitioning online. In *BIRTE*, pages 65–80, 2011.

[33] A. Jindal, E. Palatinus, V. Pavlov, and J. Dittrich. A comparison of knives for bread slicing. *PVLDB*, 6(6):361–372, 2013.

[34] A. Jindal, J. Quiané-Ruiz, and J. Dittrich. Trojan data layouts: right shoes for a running elephant. In *SOCC*, page 21, 2011.

[35] S. Kirkpatrick et al. Optimization by simmulated annealing. *Science*, 220(4598):671–680, 1983.

[36] Y. Li and J. M. Patel. Widetable: An accelerator for analytical data processing. *PVLDB*, 7(10):907–918, 2014.

[37] T. W. Manikas and J. T. Cain. Genetic algorithms vs. simulated annealing: A comparison of approaches for solving the circuit partitioning problem. 1996.

[38] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database design. *TODS*, 9(4):680–710, 1984.

[39] Y. Nourani and B. Andresen. A comparison of simulated annealing cooling strategies. *Journal of Physics A: Mathematical and General*, 31(41):8373, 1998.

[40] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[41] S. Papadomanolakis and A. Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In *SSDBM*, pages 383–392, 2004.

[42] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, 1994.

[43] J. Shafer, S. Rixner, and A. Cox. The hadoop distributed filesystem: Balancing portability and performance. In *ISPASS*, pages 122–133, 2010.

[44] D. ŚlÈľzak, J. Wróblewski, V. Eastwood, and P. Synak. Brighthouse: an analytic data warehouse for ad-hoc queries. *PVLDB*, 1(2):1337–1345, 2008.

[45] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[46] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained partitioning for aggressive data skipping. In *SIGMOD*, pages 1115–1126, 2014.

[47] H. Szu and R. Hartley. Fast simulated annealing. *Physics letters A*, 122(3):157–162, 1987.

[48] 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*, pages 996–1005, 2010.

[49] R. Van Meter. Observing the effects of multi-zone disks. In *The Usenix Technical Conference*, pages 19–30, 1997.

[50] T. White. The hadoop distributed file system. In *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 4 edition, March 2015.

[51] D. Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[52] Y. Yan, L. J. Chen, and Z. Zhang. Error-bounded sampling for analytics on big sparse data. *PVLDB*, 7(13):1508–1519, 2014.

[53] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, page 95, 2010.

[54] J. Zhou, N. Bruno, and W. Lin. Advanced partitioning techniques for massively distributed computation. In *SIGMOD*, pages 13–24, 2012.

APPENDIX

A. DISK CHARACTERS AND SEEK COST

A hard disk drive (HDD) is physically comprised of a number of platters vertically aligned together. On top of each platter there is a head reading from and writing to the platter surface. A head reads/writes one circular path at a time as the platter spins, and each circular path is referred to as a *track*. All heads are mounted on an actuator arm, and as a result all heads move together over the platters' tracks with the same radius. The tracks with the same radius are commonly referred to as a *cylinder*. A disk seek is a movement of the arm (and all the heads) from one cylinder to another. Hence, a disk seek occurs when we read two pieces of data from two cylinders.

When data is written to a new file through the OS API, OS will continuously requests disk pages one after another until all data is persisted. From the disk side, the disk controller always allocates "the next page" to accommodate the continuous page requests for creating a file. While the actual location of "the next page" may vary, depending on the location of last page, allocation follows the following precedence: a disk page on the same track of the same platter, a page on a different platter but in the same cylinder, and finally a page on an adjacent track in the current or a different platter (i.e., an adjacent cylinder). This precedence ensures that (1) a file will be placed in one cylinder if it fits, resulting in no disk seek when reading the file, and (2) when the file size exceeds the capacity of one cylinder, the file will be placed in consecutive cylinders. Therefore, the seek cost of reading two pieces of data is highly related to the number of cylinders they are apart.

In hardware community, seek cost is generally modeled as a function of the distance in number of cylinders. In particular:

1) For very short seeks (say less than 2-4 cylinders), the seek cost is dominated by the settle time of arm [42].

2) For short seeks (say less than 200-400 cylinders), the arm accelerates up to the halfway point in a constant-acceleration phase, then decelerates and settles, given a seek cost of the form $t = a + b\sqrt{s}$, where s is the seek distance in cylinders [42, 31].

3) For long seeks (say more than 200-400 cylinders), the arm reaches and moves at its maximum velocity. The seek cost is a linear function of the distance in cylinders[42, 31, 22].

As in a file system, we are willing to know the seek cost from one offset to another, i.e. in the distance of bytes, we have to consider the capacity of cylinders been skipped by a seek. In fact, cylinder capacity varies as radius changes. Cylinders in a disk are divided into a number of zones [22, 49]. Cylinders in the outside zone have large capacity than those in the inside zones. The capacity of a zone is generally quite large (in GBs) [49]. Within the same zone, seek distance in bytes is linear to the distance in cylinders. After seeking to the target cylinder, the reader has to wait the right sector been rotated under the head. So that disk rotation latency is also included in the cost of a seek operation in a file system.

With these disk characters, we can build a seek cost function of the seek distance. In practice, however, many detailed parameters of disks are impossible to obtain. Disk manufacturers usually show only the average seek time or min/max seek time in their manuals. Moreover, hidden details in file systems / operating systems between file reader and disks may also influence the seek cost. This is why we build the seek cost model by empirical approach in this paper.

B. PROOF OF THEOREM 1

PROOF. We first introduce the classic Hamilton Path Problem (HPP[7]): given an undirected graph G with n nodes and m edges, decide if there exists a path that traverses each node exactly once. We prove Theorem 1 by reducing the HPP problem to the decision version of the Column Ordering Problem (DCOP): decide if $Cost(Q, S^*)$ is less than or equal to a given number K.

Given an HPP instance, we construct a DCOP instance as follows. We first remove any duplicated edges or self-loops in the graph G. For each node x, we construct a column C_x with an arbitrary positive size. For each edge connecting two nodes x and y, we construct a query with weight 1 which will access columns C_x and C_y. We define the f function mentioned in Definition 2 as:

$$f(dist) = \begin{cases} 0, & dist = 0 \\ 1, & dist > 0 \end{cases}$$

Then we set K in the DCOP to $m - n + 1$.

Next, we prove that these two instances are equivalent. Note that the seek cost of a query in this DCOP instance is 0 if the two columns it accesses are adjacent in the column ordering and 1 otherwise. Because we removed duplicated edges, the lower bound of $Cost(Q, S^*)$ will be $m - n + 1$ because there are only $n - 1$ adjacent pairs in any ordering. Thus, it is clear that $Cost(Q, S^*)$ is less than or equal to $m - n + 1$ if and only if there exists a column ordering such that each of its $n - 1$ adjacent pairs is accessed by a query, which is equivalent to a Hamilton Path in the HPP instance.

Clearly, we reduce the Hamilton Path Problem to the desicion version of our Column Ordering Problem in polynomial time. Therefore, the Column Ordering Problem is NP-Hard. □

C. PARAMETER SETTINGS IN SCOA

SCOA has three important parameters: the initial temperature t_0, the $cooling\_rate$ and the maximum number of iterations k_{max}. These parameters are set following the best practices of SA. Parameters t_0 and $cooling\_rate$ have been studied in SA literature [25, 39, 47, 35]. The impacts of t_0 and $cooling\_rate$ are shown in Table 5. Considering that the SCOA algorithm works in an offline manner, the quality of the result is far more important than the algorithm's running time. As suggested in [25, 39], we set t_0 to be slightly larger than the seek cost (energy) of S_0.

The parameter k_{max} controls the number of iterations executed in the main loop (Line 2) of Algorithm 1. We do not apply the seek cost (energy) of candidate column orders (states) as a terminating condition because the seek cost varies significantly for various workloads, which makes it hard to set an appropriate termination threshold. The final temperature at the end of algorithm execution is $t_0 * (1 - cooling\_rate)^{k_{max}}$. So k_{max} and $cooling\_rate$ are set to ensure that the temperature can be cooled down to near zero within a given computational time because the algorithm is more likely to be trapped in a local minimum during the stage of lower temperature. As t approaches zero, the algorithm is unlikely to transit to a worse candidate state with a higher seek cost, which finally leads to convergence.

Table 5: The impacts of the initial temperature and cooling rate in SA

	Running time	Optimisation
Initial temperature ↑	↑	↑
Initial temperature ↓	↓	↓
Cooling rate ↑	↓	↓
Cooling rate ↓	↑	↑

D. COLUMN ORDERING BASED ON GENETIC ALGORITHM

Genetic Algorithm (GA) is another well-known meta-heuristic to solve hard optimization problems [51]. It is an iterative procedure that attempts to mimic evolution by maintaining a population of candidate solutions (individuals). Each individual is represented by a data structure called chromosome. In each iteration (called a generation), individuals are evaluated by the fitness of their chromosomes. Best fit individuals in a generation are selected as parents by a selection process. Then, the selected individuals are used to generate children for the next generation by means of crossover and mutation. After a computation budget is exhausted, GA returns an viable solution.

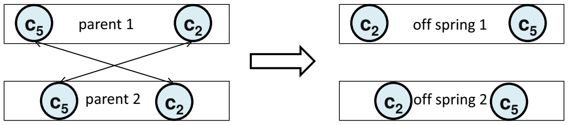


Figure 18: The crossover function in a GA-based column ordering algorithm

In the GA-based column ordering algorithm, each column order is defined as an individual. The fitness function is the seek cost function. In each generation, a number of column orders are evaluated, and the fittest ones among them are selected to generate the offspring. To do so, in the crossover function, two parents are crossed by randomly swapping a pair of columns (Figure 18). Each offspring column order is then mutated by randomly swapping two columns. Such crossover and mutation functions are similar to the random swapping method used in SCOA to generate the neighbour column order.

We implemented both SA and GA based column ordering algorithms and compared their performance. We tried our best to optimize both implementations. As shown in Figure 19, we found that the GA-based algorithm quickly got trapped in a local minimum while the SA-based algorithm always converged to much better solutions.

In both GA and SA, we further evaluated different heuristic rules, such as putting two columns which are frequently accessed in the same queries together instead of swapping two randomly chosen columns. However, it turns out that none of the adapted heuristics we tried helps. On the contrary, they make both algorithms get trapped in local minima faster and finally output worse column orders. Ultimately, in our evaluations, the SA-based algorithm with random neighbour generation works better and this is why we decided to focus on this algorithm in the paper.

E. COLUMN ORDERING BASED ON AUTOPART

As far as we know, there is no existing research work on solving column ordering and duplication problem for big wide tables. However, as discussed in Section 7, there are highly related works on vertical data partitioning. In this paper, we take the AutoPart algorithm in [41] as prototype in designing a baseline column ordering algorithm named AutoPart-C in this paper. AutoPart is evaluated to be very effective in boosting I/O performance of disk-based analytical system [33].

AutoPart partitions the data in two steps: 1) horizontally partitions the table by the categorical information of queries such that each partition is accessed by a different subset of queries. 2) vertically partitions each horizontal partition so that queries do not need to read unnecessary attributes (columns). In step 2, Auto part firstly generates the set of atomic fragments (partitions). A vertical partition is atomic there are no queries access a subset of attributes in it. Data in a fragment is stored as row-wise. I/O overhead on atomic fragments is considered to be minimal [41]. Thereafter, to reduce joining cost, in each iteration of a loop, the fragments are combined into a set of composite fragments. Attributes may be replicated among multiple fragments.

To apply AutoPart in column ordering, we need a few points of modifications: 1) as our workload do not have the 'false sharing' (query do not actually share the same horizontal partitions) features, we do not need the first horizontal partitioning stage; 2) as we are solving column ordering problem, we need to ensure there is no replications of columns; 3) since seek cost is proven to be the main overhead for our workloads and reducing seek cost is the goal of column ordering, we need to replace the cost model in AutoPart with ours.

For point 2 above, we consider using the partition merging strategy in HillClimb in [28]. HillClimb focuses on cache-efficient attribute layout within a data page. It starts from pure column layout. In each iteration of a loop, the algorithm finds and merges two vertical partitions which provide the most significant gain when merged. There is no column replicated in HillClimb.

Algorithm 4: AutoPart-C

```
   Input: The set of queries $Q = \{q_1, q_2, ..., q_m\}$;
   The initial column order $S_0 = \{c_1, c_2, ..., c_n\}$
   Output: The optimized column order $S$;
1  $Q := SortByWeight(Q)$, $F := \varnothing$, $S := S_0$;
2  for each q in Q do
3      $af := AtomicFragment(q.columns)$;
4      $F.insert(af)$;
5  while $F.size > 1$ and $Cost(F.columnOrder) < Cost(S)$ do
6      $S := F.cloumnOrder$;
7      $(f_i, f_j) := getBestPair(F)$;
8      $F.remove(f_i, f_j)$;
9      $F.insert(Combine(f_i, f_j))$;
10 return $S$;
```

In AutoPart-C, we take a fragment as a subset of ordered columns. As shown in Algorithm 4, AutoPart-C firstly sorts the queries in workload Q by the weight (gives priority to query with higher weight) and then generates the initial list of fragments F (line

2-4). $AtomicFragment$ function maintains the set of columns processed by it and generates the atomic fragment by removing processed columns from q's accessed column set. $insert$ function of F inserts a fragment into F at the best location (with maximum gain). After that, at each iteration of a loop (line 5-9), AutoPart-C finds and combines two fragments which provide the most significant gain when combined. Duplicated fragments are removed in line 8. The algorithm will converge when the workload's seek cost on F can not be further reduced or there is only one fragment in F.

Taking our workload and table schema in this paper as input. AutoPart-C generates 64 atomic fragments and takes 17 iterations to converge. Each iteration runs in $O(|S_0| \cdot |Q| \cdot |F|^3)$, so that AutoPart runs very slowly, takes about three hours to converge in our evaluations. While our SCOA converges in a few minutes.

Actually, we also considered applying AutoPart in column duplication. We remove the $F.remove$ statement (line 8) from AutoPart-C. But the algorithm converges in a few iterations without providing any gain over AutoPart-C.

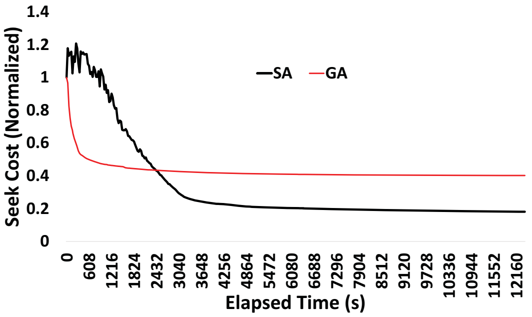


Figure 19: Performance comparison: GA-based algorithm vs. SA-based algorithm

F. DATA LAYOUT UPDATING

In our adaptive data layout optimization solution, new data layout is applied only to the incoming data, without reloading the historical data. The reasons are: 1) In most analytical workloads, historical data may be very large and it is very expensive to reload the historical data. 2) It is common that queries in log analytical workload are executed periodically (e.g. daily or weekly) for generating reports and building data caches, so most of the queries access recent data and rarely scan long-term historical data. It is therefore most important for us to optimize the column layout of the new coming data which is frequently accessed by the current workload.

But it is possible that a query accesses both old and new data when the new column layout is applied. This may cost problems to query execution. Column duplication is currently a separate and optional feature in our solution, so we discuss this problem for column ordering and duplication separately:

1. Column ordering layout (without column duplication): Queries can access both old and new data without any efforts in modifying or configuring the existing storage or data analytical systems. During query execution, the access order of the columns is determined by the metadata inside of each row group and is transparent to the data analytical system. Take Parquet as an example, the set of columns to be read in the query plan is structured as a HashSet without order. This set of columns are mapped to the data schema store in the metadata footer of each Parquet file (each Parquet file contains only one HDFS block as recommended by [3]). So SQL queries can access the row groups of different column order without even noticing the difference. Further more, legacy queries in the workload are not likely to have worse performance on both old and new data. In Figure 10, the minimal gain of a query of *SCOA* over *Default* in the workload is 0.9%, which means that not a single query is punished by column ordering. Since legacy queries are part of the both old and new workload which are used to produce the old and new column ordering layout, they are not likely to have worse performance (compared with the query performance on default column order) on both old or new data.

2. Column duplication. For queries which access both old and new data, duplication is not used. These queries are directed to the column ordering layout (without column duplication). In production, most of the jobs are recurring jobs running daily or weekly while data layouts are generally updated monthly. Therefore, in most cases, queries only access new data and column duplication is applicable. To address the problem of accessing both old and new data on a column duplication layout, we need to make changes to the data analytical systems. We are in the process of developing such changes inside Spark. Data with different duplication layouts will be handled in the map stage (doing table scan and filter) of the Spark query plan and is transparent to applications.