Trojan Data Layouts: Right Shoes for a Running Elephant*

Alekh Jindal Jorge-Arnulfo Quiané-Ruiz Jens Dittrich

Information Systems Group
Saarland University
http://infosys.cs.uni-saarland.de

ABSTRACT

MapReduce is becoming ubiquitous in large-scale data analysis. Several recent works have shown that the performance of Hadoop MapReduce could be improved, for instance, by creating indexes in a non-invasive manner. However, they ignore the impact of the data layout used inside data blocks of Hadoop Distributed File System (HDFS). In this paper, we analyze different data layouts in detail in the context of MapReduce and argue that Row, Column, and PAX layouts can lead to poor system performance. We propose a new data layout, coined Trojan Layout, that internally organizes data blocks into attribute groups according to the workload in order to improve data access times. A salient feature of Trojan Layout is that it fully preserves the fault-tolerance properties of MapReduce. We implement our Trojan Layout idea in HDFS 0.20.3 and call the resulting system Trojan HDFS. We exploit the fact that HDFS stores multiple replicas of each data block on different computing nodes. Trojan HDFS automatically creates a different Trojan Layout per replica to better fit the workload. As a result, we are able to schedule incoming MapReduce jobs to data block replicas with the most suitable Trojan Layout. We evaluate our approach using three real-world workloads. We compare Trojan Layouts against Hadoop using Row and PAX layouts. The results demonstrate that Trojan Layout allows MapReduce jobs to read their input data up to 4.8 times faster than Row layout and up to 3.5 times faster than PAX layout.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

General Terms

Design, Performance

Keywords

MapReduce, per-replica data layout, column grouping

*Work partially supported by the Saarbrücken Cluster of Excellence on Multimodal Computing and Interaction (M2CI).

SOCC'11, October 27–28, 2011, Cascais, Portugal.

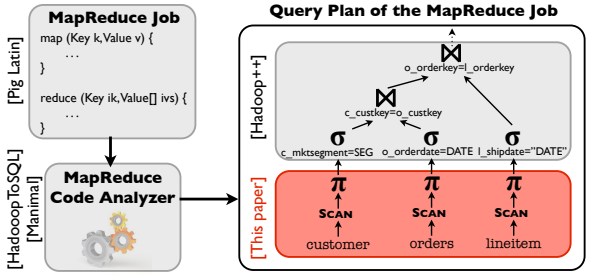


Figure 1: Example of some research works in MapReduce.

1. INTRODUCTION

Analyzing terabytes of data on a daily basis is a common task for many enterprises such as Google, Facebook, and Yahoo!. With this trend, MapReduce [13] is quickly becoming the *de facto* standard for large-scale analysis in industry. However, it has been shown that MapReduce suffers from very slow execution times in analytical queries compared to DBMSs [26]. Several recent works have improved the performance of MapReduce. Figure 1 illustrates the research focus of some of these research works. For instance, Pig Latin by Olston et al. proposed a new interface to execute MapReduce jobs [25] (top-left box in Figure 1); other researchers proposed HadoopToSQL [19] and Manimal [7] to automatically analyze the code of MapReduce jobs in order to produce more efficient query plans (bottom-left box); we recently proposed Hadoop++, a new system that improves the performance of MapReduce jobs by injecting code into the MapReduce query plans [14] (top-right box). Many other works have focused on improving MapReduce in other aspects [27, 18, 20, 22, 24]. However, none of these works considers the impact of data layouts, per distributed file system (DFS) data block, on the data read performance of MapReduce jobs. This paper fills this gap: we analyze the different data layouts in detail. The red part in Figure 1 illustrates the focus of this paper.

1.1 Background and Motivation

Traditional Layouts in MapReduce. Currently, MapReduce processes input data blocks in a strictly row-oriented fashion. Thus, all tuple attributes have to be read from disk even if only some of them are relevant to process a given task. The disadvantage of a row-oriented layout has been thoroughly researched in the context of column stores [1, 3, 9, 31, 28]. However, in a distributed system a column store has severe drawbacks as the data blocks for different columns may reside on different nodes. Thus, whenever a query references more than one attribute, columns have to be sent through the network in order to merge different attributes values into a row

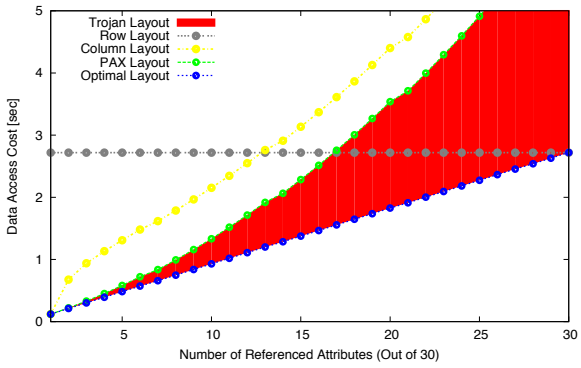


Figure 2: Data access costs for different data layouts in Hadoop.

(*tuple reconstruction*). For instance, consider a table `AccessLog` containing access log-records of a web server. Assume the following simple SQL-query:

```
SELECT url, sourceip FROM AccessLog
WHERE url LIKE '%.edu%';
```

To process this query MapReduce needs to either: (1) fetch and scan all columns, join them to reconstruct tuples containing attribute values for both `url` and `sourceip`, and then filter those tuples to only return the ones containing ".edu" in their `url`; or (2) scan column `url`, collect the tuple-IDs of matching tuples and match them to retrieve the `sourceip` value of each tuple. The latter process is called late materialization [2]. In either case, the problem is that the attributes referenced after the `SELECT` clause have to be fetched over the network in many cases. This can significantly decrease the performance of MapReduce jobs.

Hybrid Layout in MapReduce. For these reasons a clever optimization is to use a hybrid layout of columns and rows. The idea is to keep the same data on a block as we would keep in a row layout. However, in contrast to a row layout, *inside* a block data is organized into a column layout. This approach is termed PAX (partition across) and was first proposed in the context of page organization in a DBMS [6]. Recently, it was also introduced in MapReduce [11, 32]. Using PAX in MapReduce has big advantages: (1) tuple reconstruction does not trigger expensive fetches over the network, as all data values belonging to a tuple are locally available inside a block, (2) the execution pipeline does not have to be changed at all to implement complex tuple reconstructing joins, and (3) as data blocks are typically large, about 256 MB, the subpage containing data for a specific attribute is very large. For instance, assuming a table having 30 attributes of equal size, each subpage still contains 8.5 MB of data! As a result, reading a subpage amounts to a sequential scan of 8.5 MB of data, which is typically very fast on disk. To process the SQL query mentioned above, it is then sufficient to read the subpage containing data for attribute `url` only. Then, for the qualifying tuples, we only need to access data from the subpages containing the `sourceip`. For this, we may scan that subpage entirely or elevator-scan the subpage skipping some parts in the scan. In either case, we do not trigger any network requests to fetch missing attributes. All data is local *within* the data block.

Different Layout Performance in MapReduce. Figure 2 shows the comparison of the estimated access cost of running a MapReduce job on each of the Row, Column, and PAX Layouts. Additionally, we consider the Optimal Layout, which co-locates all attributes referenced by any incoming query into a single column group. We use a query referencing a single input table of 30 attributes. Our cost model considers random and sequential I/O, network, and even the scheduling decisions made by the MapReduce scheduler (see Appendix A for details of the cost model and this simulation). The results in Figure 2 show that Column Layout is not competitive compared to PAX Layout in MapReduce in a distributed setting. This is due to the high costs for fetching missing attribute values over the network as explained above. We also observe that PAX Layout is better than Row Layout for up to 17 (out of 30) referenced attributes. Beyond that, PAX Layout is worse than Row Layout, because the number of individual seeks in PAX adds considerable random I/O due to buffered reads of the individual subpages. In addition, tuple reconstruction in PAX adds considerable CPU costs. Even if PAX Layout seems to perform well in many cases, we observe that there exists a big gap between PAX Layout and Optimal Layout. This is because the Optimal Layout always groups all referenced attributes together, thereby requiring fewer seeks and no tuple reconstruction. Therefore, it is quite important for the performance of applications to pack as many referenced attributes as possible co-located together.

1.2 Our Approach and Research Challenges

In this paper, we propose a new approach coined *Trojan Layout*. Like PAX, Trojan Layout keeps the same data inside a block. However, in contrast to PAX, we allow for any internal data layout inside a block. The possible improvement of Trojan Layouts over PAX is depicted by the red space in Figure 2. Interestingly, we already see an improvement of ~270% over Column Layout and of ~20% over Pax Layout for five referenced attributes.

Additionally, we exploit the existing data block replication in Hadoop DFS (HDFS) to create different Trojan Layouts on a per-replica basis. This means that rather than keeping all data block replicas in the same layout, we use different Trojan Layouts for each replica. Each replica is optimized for a different subclass of queries. As a result, every incoming query can be scheduled to the most suitable data block replica. In a special case this would efficiently mimic fractured mirrors [28], which maintain two copies of the data: one in Row Layout and other in Column Layout. The reader may think that this is also possible in HDFS by using pure Row and Column Layouts. However, doing so would significantly impact the fault-tolerance properties of HDFS, because a data block replica would not contain the same data in Row Layout as in Column Layout. Therefore, complex mechanisms would be required to identify, track, and reconstruct lost data block replicas.

The idea of Trojan Layouts triggers a number of interesting research challenges. First, we have to cluster a given workload into query groups based on their access pattern in order to better exploit different data block replicas. Second, we need to invent efficient algorithms to determine the right Trojan Layout for each data block replica. Although some existing work from vertical partitioning may be leveraged [16, 5, 17], these algorithms have issues. They have to be extended to (i) improve the quality of vertical partitioning, and (ii) support replicas of the same block in different layouts. Third, we should not force users into manually defining data block layouts. If we did that, we might eventually end up turning MapReduce into yet another DBMS, with a few hundred different knobs to be properly set by a skilled (and expensive) database administrator. However, the ease-of-use and the low administration costs of MapReduce are some of its biggest advantages over DBMSs.

Therefore, the main problem we tackle in this paper is as follows. Given an incoming query workload, we have to determine the right Trojan Layout for each data block replica that: (i) approaches to optimal layouts in performance, (ii) keeps the interface of MapReduce intact, and (iii) is zero-admin, which is extremely important

for future distributed systems as emphasized in the conclusion section of the ten-year best paper award of Surajit Chaudhuri [10].

1.3 Contributions

Trojan Layouts are inspired by PAX in the sense that we only change the internal organization of a data block and not among data blocks. However, we considerably depart from PAX as we can: (i) co-locate attributes together according to query workloads, (ii) use different Trojan Layouts for different data block replicas, and (iii) in a special case, mimic fractured mirrors: having the best from both PAX and Row Layouts. In summary, we make the following key contributions:

1. We propose a column grouping algorithm in which we first (i) determine column groups using a novel interestingness measure, which denotes how well a set of attributes speeds up most or all queries in a workload; and then (ii) pack the column groups in order to maximize the total interestingness of data blocks. We use this algorithm as a basis to determine the Trojan Layout of data blocks in HDFS. It is worth noting that even if we focus on MapReduce in this paper, one can use our column grouping algorithm in other domains as well.

2. We exploit default HDFS data replication to create a different Trojan Layout per data block replica. For this, we first show how to apply our column grouping algorithm for query grouping as well, i.e. for clustering queries in a workload according to their access patterns. We then map each resulting query group to one data block replica so as to compute the Trojan Layout for such a replica.

3. We present Trojan HDFS, a (per-replica) Trojan Layout aware HDFS. At data upload time, Trojan HDFS automatically transforms data block replicas into their corresponding Trojan Layouts; it hides all messy details from the user. Thereafter, Trojan HDFS keeps track of Trojan Layouts for each data block replica. With Trojan HDFS, neither the MapReduce processing pipeline nor the MapReduce interface are changed at all.

4. We evaluate Trojan Layouts using three real-world workloads: TPC-H, Star Schema Benchmark (SSB), and Sloan Digital Sky Survey (SDSS). The results demonstrate that Trojan Layouts allow MapReduce jobs to read data up to a factor of 4.8 faster than Row Layout and up to a factor of 3.5 faster than PAX Layout.

2. OVERVIEW

We propose Trojan Layouts as our solution to decrease the waiting time of data-intensive jobs when accessing data from HDFS. The core idea of Trojan Layouts is to internally organize data blocks into column groups according to the workload. Our approach has three phases: (1) *compute* the Trojan Layout for each data block replica, (2) *create* the computed Trojan Layouts in HDFS, and (3) *access* the existing Trojan Layouts. From the user perspective, the data analysis workflow remains the same: upload the input data and run the query workload exactly as before.

Given a query workload W, at upload time we determine the Trojan Layout for each data block replica. We then store each data block replica in its respective Trojan Layout. We illustrate the core idea of Trojan Layouts in Figure 3. A data block using a Trojan Layout is composed of `Header` metadata and a set of column groups (see Replica 1 in data node 1). The header contains the number of attributes stored in a data block and attribute pointers.

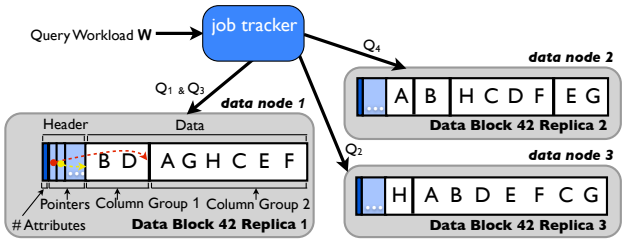


Figure 3: Per-replica Trojan Layouts in HDFS.

An attribute pointer points to the beginning of the column group that contains that attribute. For instance, in Replica 1, the first attribute pointer (the red arrow) points to Column Group 2, which contains attribute A. The second attribute pointer would then point to Column Group 1, and so on. Each column group in turn contains a set of attributes in row-fashion, e.g. Column Group 1 in Replica 1 has tuples containing attributes B and D.

At query time, we transparently adapt an incoming MapReduce job to query the data block replica that minimizes the data access time. Then, we route the map tasks of the MapReduce job to the data nodes storing such data block replicas. For example, in Figure 3, the map tasks of Q_4 are routed to data node 1, while the map tasks of Q_1 and Q_3 are routed to data node 2 and those of Q_2 are routed to data node 3. Notice that in case the scheduler cannot route a map task to the best data block replica, e.g. because the data node storing such a replica is busy, the scheduler transparently fallbacks to other Trojan Layouts. An important feature of our approach is that we keep the Hadoop MapReduce interface intact by providing the right *itemize* function in the Hadoop plan [14]. As in normal MapReduce, users only care about `map` and `reduce` functions.

The salient features of our approach are as follows:

- *Invisibility*. We create and access Trojan Layouts in a way that is invisible to users.

- *Non-Invasive*. We do not change the outside HDFS unit, i.e. the data block, but rather change the internal representation of a data block. As a result, we do not require any change in the MapReduce processing pipeline for accessing Trojan Layouts.

- *Seamless Query Processing*. We seamlessly wrap the input data to a given MapReduce job. Obviously, we do this for the data block replica that minimizes the time for reading required data.

- *Rich DBMS features support*. One can easily enrich Trojan Layouts with standard DBMS optimizations, such as indexing, partitioning, and co-partitioning.

- *Per-Replica Trojan Layout*: We exploit existing data block replication in HDFS to create a different Trojan Layout for each replica so as to better fit to the workload.

We provide the details on how we compute Trojan Layouts in Section 3. Then, in Section 4, we describe how we enable HDFS to store each data block replica in a different Trojan Layout in Trojan HDFS. In the same Section 4, we discuss how we physically create Trojan Layouts. In addition, we discuss the query processing and scheduling aspects of our approach.

3. INTERESTINGNESS-BASED COLUMN GROUPING ALGORITHM

The core idea of Trojan layouts is to adapt the internal representation of data blocks, while the outside view of data for the rest of the data processing pipeline remains the same. This speeds up query execution. Since we focus on scan and projection operators in the query plan, we restrict ourselves to attribute level data adaptation, i.e. group sets of attributes together. Given a relation with attribute set $\mathbb{A}$, we consider a column group $G \subseteq \mathbb{A}$ as any subset of $\mathbb{A}$. To understand the intuition behind column grouping, let us consider the queries and their access pattern in Example 1 below.

EXAMPLE 1. *Access pattern of attributes A,B,C,D in queries Q_1–Q_{10}.*

	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}
A	1	1	1	1	0	0	0	0	0	0
B	1	1	1	1	0	0	0	0	0	0
C	0	0	0	0	1	1	1	1	1	1
D	0	1	1	1	1	1	1	1	1	1

In the above table, if a query accesses an attribute, then the corresponding cell has value 1, otherwise it has value 0. Notice that attributes A and B are co-accessed in queries Q_1 to Q_4 (■ in Q_1 to Q_4 of Example 1). Thus, column group {A,B} is interesting as it can speedup queries Q_1 to Q_4. As another example, consider the queries and their access pattern in Example 2 below.

EXAMPLE 2. *Access pattern of attributes M,N,O,P in queries Q_{11}–Q_{20}.*

	Q_{11}	Q_{12}	Q_{13}	Q_{14}	Q_{15}	Q_{16}	Q_{17}	Q_{18}	Q_{19}	Q_{20}
M	1	1	0	0	0	0	0	1	1	0
N	1	1	1	0	0	0	0	1	1	1
O	0	1	1	0	0	0	0	0	1	1
P	1	1	0	1	1	1	1	0	0	0

Observe that attributes M,N and N,O are co-accessed respectively in queries $\{Q_{11},Q_{12},Q_{18},Q_{19}\}$ and $\{Q_{12},Q_{13},Q_{19},Q_{20}\}$ (■ in Example 2). This makes column group {M,N,O} an interesting one. Thus, generally speaking, a column group is interesting if pairs of attributes in the column group are co-accessed (e.g. M,N and N,O in Example 2), even though all attributes in the column group may not be co-accessed. Consequently, in order to find the most suitable internal representation of data in data blocks, we focus on two core operations: (i) determining the interesting column groups, and (ii) packing them within a data block such that the total *interestingness* of the data block is maximized. Denoting a set of complete and disjoint column groups as $\mathbb{G}'$, where $\mathbb{G}'$ is a subset of the set of all possible column groups $\mathbb{G}$, we can now describe the problem we address as follows:

Problem Statement. *Given a column group interestingness function $Intg(G) \rightarrow [0,1]$, find the complete and disjoint column group set $\mathbb{G}'$ that maximizes the total interestingness of a data block*, i.e. max $(\sum_{\forall G \in \mathbb{G}'} Intg(G))$.

To approach the above problem, we first describe our novel *column group interestingness* function and compare its effectiveness with prior approaches below. Thereafter, we map the packing of column groups, which is an NP-hard problem [29], to a 0-1 Knapsack problem and solve it using a *branch and bound* technique.

3.1 Column Group Interestingness

Intuitively, a column group is highly interesting if it speeds up most or all of the queries in the workload. Thus, to formally define the interestingness of a column group, we first consider the access costs of queries in query workload W. Let Path(`Opt`, Q) denote the access path chosen by optimizer `Opt` for query Q and let $B_A(Q, \text{Path}(\texttt{Opt}, Q))$ denote the number of bytes of attribute A read by query Q when using access path Path(`Opt`, Q)[1]. We denote the total bytes consumed by a query Q as its *footprint* F_Q:

$$F_Q = \sum_{A \in \mathbb{A}} B_A(Q, \text{Path}(\texttt{Opt}, Q)).$$

Let us now understand which attributes contribute to the query footprint. For this, traditionally e.g. [5], one would use an attribute usage matrix $U(Q,A)$ to indicate whether or not an attribute A is referenced by query Q, i.e U(Q,A)=1, if Q references A, and 0 otherwise. However, $U(Q,A)$ considers only the attribute occurrences (■ in Example 1), even though attribute *non-occurrences* give equally important information: they are crucial in determining whether one attribute should co-occur with another or not. For instance, in Example 1, attributes C and D have common non-occurrence only in query Q_1 whereas for queries Q_2–Q_4 column group {C,D} will have redundant access of C (■). In contrast, attributes A and B have all non-occurrence in common (queries Q_5–Q_{10}) and therefore column group {A,B} is more interesting. To capture this, we generalize $U(Q,A)$ using a binary variable x, which denotes the occurrence ($x = 1$) and the non-occurrence ($x = 0$) of an attribute.

$$U_x(Q,A) = \begin{cases} U(Q,A) & \text{if x=1,} \\ 1 - U(Q,A) & \text{if x=0.} \end{cases}$$

Notice that the above usage matrix does not take into account the footprints (total byte access) of queries in which they occur.

EXAMPLE 3. *footprint and attribute usage in queries Q_1 to Q_4.*

	Query footprint
$\mathbf{Q_1}$	10
$\mathbf{Q_2}$	20
$\mathbf{Q_3}$	30
$\mathbf{Q_4}$	40

	Q_1	Q_2	Q_3	Q_4
A	1	1	0	0
B	0	0	1	1
C	0	1	0	1

For instance, in Example 3, attributes A,B,C have the same frequency in the workload. However, attributes A,C are co-referenced by the cheaper query Q_2 (i.e. having smaller footprint) whereas attributes B,C are co-referenced by more expensive query Q_4 (i.e. having bigger footprint), thereby making B,C more likely to be together. Therefore, we introduce the *relative importance* (*RI*) of attributes, which takes query footprints into account. Intuitively, RI_A is the fractional reading cost in the events when an attribute A occurs as well as when it does not. We define RI_A as follows:

$$RI_A(x) = \frac{\sum_{Q \in W} F_Q \cdot U_x(Q,A)}{\sum_{Q \in W} F_Q}.$$

RI_A is normalized by the total workload costs to make it comparable. Since we want to co-locate attributes inside data blocks, we need to determine whether two attributes should be stored together. Thus, we also define $RI_{A,B}(x,y)$ as the *relative importance of an attribute pair A, B* in terms of the query workload cost.

$$RI_{A,B}(x,y) = \frac{\sum_{Q \in W} F_Q \cdot U_x(Q,A) \cdot U_y(Q,B)}{\sum_{Q \in W} F_Q}.$$

[1]The choice of the access path depends on the optimizer, which can choose either the index access path or the table scan access path. For instance, the optimizer can come up with the access path reading lesser number of bytes. Without any loss of generality, one can supply other Path(`Opt`, Q) functions to our algorithm.

Now, to estimate the similarity between two attributes A and B over the range of values of x and y, we measure their mutual dependence using the mutual information [21] between them. We can compute the *mutual information between two attributes* using their relative importances as follows:

$$MI(A,B) = \sum_{x\in\{0,1\}} \sum_{y\in\{0,1\}} RI_{A,B}(x,y) \cdot \log\left(\frac{RI_{A,B}(x,y)}{RI_A(x) \cdot RI_B(y)}\right).$$

Essentially, $MI(A,B)$ measures the information (data access patterns) that attributes A and B share. We normalize $MI(A,B)$ by the minimum entropies of the two attributes to normalize its range between 0 and 1, i.e. $nMI(A,B) = \frac{MI(A,B)}{min(H(A),H(B))}$. Here, $H(A)$ and $H(B)$ denote the entropy of attributes A and B. For an attribute A, we compute its entropy as: $H(A) = \sum_{x\in\{0,1\}} RI_A(x) \cdot \log\left(\frac{1}{RI_A(x)}\right)$. Finally, we can define column group interestingness.

Definition 1. **Column Group Interestingness** *of a column group G is the average normalized mutual information of any given attribute pair in G. Formally,*

$$Intg(G) = \begin{cases} \frac{1}{\binom{|G|}{2}} \cdot \sum\limits_{\{A,B\}\in G, A\neq B} nMI(A,B) & |G| > 1, \\ \frac{1}{|\mathbb{A}|-1} \cdot \sum\limits_{A\in G, B\in\mathbb{A}\setminus G} 1 - nMI(A,B) & |G| = 1. \end{cases}$$

■

Note that for column groups having a single attribute, we take the inverse of the mutual information with any other attribute in $\mathbb{A}$. In other words, we measure the benefit of the attribute in the column group not occurring with any other attribute in $\mathbb{A}$. $Intg(G)$ has values between 0 and 1. Higher interestingness indicates higher mutual dependence within a column group.

By default, we would have to consider all column groups ($O(2^{|\mathbb{A}|})$) within a data block. In practice, we use the similar pruning method as in [5] in order to reduce the search space. We experimentally determine the threshold interestingness value and discard all column groups having interestingness below that threshold. A higher interestingness threshold produces a smaller set of candidate column groups. This has two consequences: (i) the search space for finding the best combination of column groups (introduced as column group packing in Section 3.2) becomes smaller, and (ii) only the attributes appearing in highly interesting column groups remain in the candidate set and are thus likely to be grouped. All remaining attributes which do not appear in any of the highly interesting column groups will end up in row layout. Apart from threshold based pruning, we can perform further aggressive pruning, for column groups having same interestingness value, in two ways: (i) keep the smallest column group to reduce redundant data read, or (ii) keep the largest column group to reduce tuple reconstruction costs.

Comparison with CG-Cost [5]. It is important to note that, in contrast to [5], our definition of interestingness produces superior interestingness measure, which we illustrate as follows. The algorithm in [5] computes the interestingness (CG-Cost) for column groups {A,B} and {C,D} in Example 1 as 0.4 and 0.6 respectively. Our algorithm computes interestingness (Intg) as 1.0 and 0.23 respectively, which makes much more sense since A and B always occur/not-occur together. Likewise, the algorithm in [5] computes the interestingness for both column groups {M,N,O} and {M,P} in Example 2 as 0.2. Our algorithm computes interestingness (Intg) as 0.278 and 0.005 respectively. Again, this makes more sense since {M,N} and {N,O} are pairwise similar making group {M,N,O} more interesting.

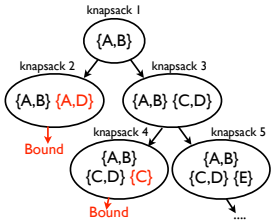


Figure 4: Branch and Bound

3.2 Column Group Packing as 0-1 Knapsack Problem

Once we have the candidate column groups along with their interestingness values, our goal now is to pack these column groups into a data block such that the total interestingness of all column groups in the data block is maximized. As mentioned before, this is an NP-hard problem [29]. Thus, we map it to a 0-1 knapsack problem, with an extra disjointness constraint, to solve it.

For a given column group G, let $id(G)$ denote the group identifier (a numeric in binary representation) such that its i^{th} bit is set to 1 if G contains attribute i, it is set to 0 otherwise. Given m column groups, we have to find 0-1 variables $x_1, x_2, ..., x_m$ — where x_i is 1 if column group G_i is selected and 0 otherwise — such that the total interestingness is maximized. Additionally, the sum of the group identifiers should be at most $id(\mathbb{A})$ and each of the groups should be *disjoint*. Formally, $max \sum_{i=1}^{m} Intg(G_i) \cdot x_i$ subject to:

$$\sum_{i=1}^{m} id(G_i) \cdot x_i \leq id(\mathbb{A}) \tag{1}$$

$$x_i + x_j \leq 1, \quad \forall i,j \text{ s.t. } i \neq j \wedge G_i \cap G_j \neq \emptyset. \tag{2}$$

Here, (2) is an extra constraint to the standard 0-1 knapsack problem. Due to this additional constraint we cannot reduce the problem to a sub-problem. This is because the solution to the sub-problem may contain items which are not disjoint in the main problem. Thus, we cannot use a dynamic programming algorithm to solve this problem. However, constraint (2) allows us to pre-filter non-disjoint column groups. Therefore, we can apply a *branch and bound* technique. The idea is to consider a column group and its subsequent combinations with other column groups, only if it is disjoint with the column groups currently in the knapsack. Figure 4 illustrates this idea. We observe that column groups {A,D} and {C} bound any further branching of knapsack iterations. Algorithm 1 shows the pseudo-code of this technique. The algorithm denotes a column group as a knapsack item, its interestingness as the benefit, and its group identifier as weight. In case we have explored all knapsack items, we check if we have a knapsack with greater benefit than before (Lines 1-10). Else, we recursively call `CG.bbKnapsack` in two cases: (i) without taking the current item into the knapsack (Line 12), and (ii) taking the current item if it satisfies constraints (1) and (2) (Lines 13–15).

It is worth noting that our interestingness function does not consider the size of the column group. However, for operators such as joins, the number and sizes of column groups would be quite important. Thus, we solve the above problem each for the number of column groups ranging from 1 to $|\mathbb{A}|$, as shown in Algorithm 2. We first generate the column groups (Line 1) and add them to the item list (Line 2), then we set the weight (group identifier) and benefit (interestingness) of each item (Lines 4–8). We set `maxWeight` to the maximum item weight and call `CG.bbKnapsack`, which returns a column group set each for the number of groups ranging from 1

Algorithm 1: Branch And Bound Knapsack:`CGA.bbKnapsack`

```
   Input  : item, benefit, weight, weightVector, itemBitMap
   Output: Max benefit item vectors, each for #column-groups from 1 to 𝔸
 1 if EndOfItemList(item) then
 2     k = NumItems(itemBitMap);
 3     if weight < MaxWeight then
 4         k = k+1;
 5     end
 6     if k > 0 and benefit > MaxBenefit(k) then
 7         CGA.SetMaxBenefit(k,benefit);
 8         CGA.SetMaxBenefitItemBitMap(k,itemBitMap);
 9     end
10 else
11     CGA.bbKnapsack(NextItemInList(item), benefit, weight, weightVector,
       itemBitMap);
12     if (weight & ItemWeight(item))==0 and (weight + ItemWeight(item)
       ≤ MaxWeight) then
13         CGA.bbKnapsack(NextItemInList(item),
           benefit+ItemBenefit(item), weight+ItemWeight(item),
           weightVector | ItemWeight(item), itemBitMap |
           ItemVector(item));
14     end
15 end
```

Algorithm 2: `EnumerateAndGroup`

```
   Input  : Items items
   Output: Group[][] itemGroupings
 1 Group [] candidates = GetSubsets(items);
 2 CGA.SetItemList(candidates);
 3 maxWeight = 0;
 4 for i=1 to size(candidates) do
 5     maxWeight = maxWeight | (1 << i);
 6     CGA.SetItemWeight(candidates[i], v(candidates[i]));
 7     CGA.SetItemBenefit(candidates[i], I(candidates[i]));
 8 end
 9 CGA.SetMaxWeight(maxWeight);
10 Group [][] groupings = CGA.bbKnapsack(0,0,0,0,0);
11 return groupings;
```

to $|\mathbb{A}|$. As the number of solutions is equal to the number of attributes in the relation, it is now feasible to compare and pick the best partitioning using a cost model (see Appendix A).

Our column grouping algorithm, along with column group pruning, works well for several realistic datasets, e.g. for TPC-H tables (having a maximum of 16 attributes) and for SSB tables (having a maximum of 17 attributes). However, finding the right Trojan Layouts for scientific data sets (having hundreds of attributes), like SDSS, becomes a difficult task to achieve. Luckily, HDFS replicates data blocks three times by default to ensure the availability of data blocks. Thus, instead of using the same data layout for all the three replicas, we create a different Trojan Layout per replica. This divide-and-conquer approach significantly reduces the complexity of our column grouping algorithm. We describe per-replica Trojan Layout in the following section.

4. PER-REPLICA TROJAN LAYOUT

In this section, we describe our novel per-replica Trojan Layouts. The core idea of per-replica Trojan Layout is to first create query groups (using the *same* column grouping algorithm) and then create column groups for each query group separately. This serves two purposes: (i) instead of creating a single layout for the entire workload, we create multiple layouts, each specialized for a part of the workload, and (ii) query grouping can significantly decrease the number of referenced attributes for each query group, which, in turn, reduces the complexity of our column grouping algorithm. Algorithm 3 shows the pseudo-code to compute per-replica Trojan Layouts in two steps:

Algorithm 3: `PerReplicaEnumerateAndGroup`

```
   Input  : Query[] queries, Attribute[] attributes, Int replicationFactor
   Output: Group[][] perReplicaGroupings
1 Group [][] queryGroupings = EnumerateAndGroup(queries);
2 Group [] queryGrouping = queryGroupings[replicationFactor];
3 Group [][] perReplicaGroups;
4 for i=1 to size(queryGrouping) do
5     Attribute [] refAttributes = GetRef(queryGrouping[i]);
6     Group [][] attrGroupings = EnumerateAndGroup(refAttributes);
7     perReplicaGroups[i] = PickBestUsingCostModel(attrGroupings);
8 end
9 return perReplicaGroups;
```

(1.) Query Grouping. We first group queries in the workload based on their access pattern. Notice that column grouping is orthogonal to query grouping. However, two queries are similar if they access similar attributes just as two attributes are similar if they are accessed by similar queries. In that respect, query grouping, or rather partitioning, is very similar to column grouping. Therefore, we use our column grouping algorithm (Algorithm 2) for query grouping as well: we just interchange attributes with queries (Line 1). To illustrate, in the attribute usage matrix of Example 1 in Section 3, query group $\{Q_1, Q_2, Q_3\}$ has an interestingness of 0.49 whereas query group $\{Q_2, Q_3, Q_4\}$, having queries with more similar access pattern, has an interestingness of 1.0. As a result of running Algorithm 2 for query grouping, we receive a collection of query group sets that are complete and disjoint. Each query group set in the collection contains a different number of query groups. We pick the query group set having as many query groups as the replication factor (Line 2), thus mapping one query group to one data block replica. However, we can as well map one query group to multiple replicas, depending on the workload.

Recall that we perform query grouping in order to reduce the complexity of our column grouping algorithm. However, if the number of queries increase then the complexity of query grouping will increase as well. To deal with this, for large workloads, we apply query grouping recursively as follows: (i) first we (independently) group consecutive sets of p queries, (ii) then we XOR the queries in a query group to represent each query group as a single combined query, (iii) now we again (independently) group consecutive sets of p combined queries, (iv) we repeat this process till we have a single set of p or less queries. Here p denotes the maximum number of queries which can be grouped in reasonable time. Experimentally, we determine p to be less than or equal to 20.

(2.) Query Routing. Finally, for each query group, we get the referenced attributes and build column groups on them (Lines 5–6 in Algorithm 2). We pick the best column grouping among groupings of different size using a cost model[2] (Line 7).

In the remainder of this section, we discuss how we support per-replica Trojan Layouts in HDFS (Section 4.1); how we transform data blocks to a given Trojan Layout (Section 4.2); how we access Trojan Layouts in Hadoop MapReduce jobs (Section 4.3); and what scheduling policies we consider (Section 4.4).

4.1 Layout Aware Replication

We implemented a variant of HDFS, called Trojan HDFS, to introduce per-replica Trojan Layouts into HDFS. Trojan HDFS differs from HDFS in two aspects:

(1.) The name node in Trojan HDFS keeps a catalog of the Trojan Layouts of all data block replicas. Trojan HDFS exploits the fact that the name node maintains a triplet of pointers for each data

[2]See Appendix A for details of the cost model.

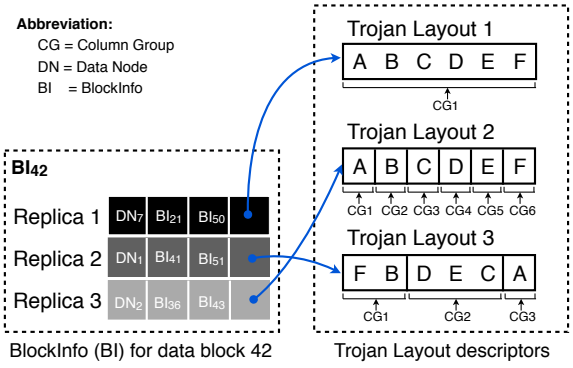


Figure 5: Quadruplets for a data block in TROJAN HDFS stored at the name node. This structured is composed: (i) of a pointer to the data node (e.g. DN 7) storing a replica (e.g. the first replica) of a data block (e.g. data block 42), (ii) of a pointer to the previous data block (e.g. data block 21) stored on DN 7, (iii) of a pointer to the next block (e.g. data block 51) stored on DN 7, and (iv) of a pointer to the Trojan Layout descriptor for that data block replica (e.g. row-layout).

block replica[3]. It adds a fourth pointer to this structure, which points to the Trojan Layout descriptor of the data block replica. Figure 5 illustrates this quadruplet of pointers associated to a data block replica. Note that more than one data block replica could point to the same Trojan Layout descriptor.

(2.) A data node in TROJAN HDFS asks the name node for the Trojan Layout of each data block replica stored locally. After receiving the Trojan Layout for a given data block replica, a data node internally reorganizes the data of the data block replica according to the received layout. There are two ways, for a data node, to do so: (i) reorganize a data block as soon as the data block replica is copied locally, or (ii) reorganize a data block after all replicas of the data block are copied to relevant data nodes. The reader might think the first strategy to be better, since data nodes do not have to wait for other replicas to be copied. However, this strategy generates contention between data nodes for accessing data block replicas. This is because a data node would be accessing a given local data block replica for transformation while another data node would be trying to remotely copy the same data block replica for replication. This contention will, in turn, significantly increase the data upload time. Therefore, in TROJAN HDFS, we apply the second strategy, i.e. data nodes start data block reorganizations after all replicas are copied. We showed an example of the resulting internal organization of a data block in Figure 3.

4.2 Layout Creation

We now focus on the process of uploading a file to TROJAN HDFS. In a spirit similar to databases physical design wizards, users have to run our *Trojan Layout Wizard* (TLW) to come up with the Trojan Layouts for their data sets. For this, users feed the TLW with the query workload, the schema of their data sets, and the replication factor they will use to store their data sets. Given these inputs, TLW computes the per-replica Trojan Layouts and returns a *layout configuration* file. The layout configuration file contains, for each data set, a row having data set name and per-replica Trojan

[3]In this triplet (i) the first pointer points to the metadata of the node storing the data block replica, (ii) the second pointer points to the previous data block stored on the same data node, and (iii) the third pointer points to the next data block stored on the same data node.

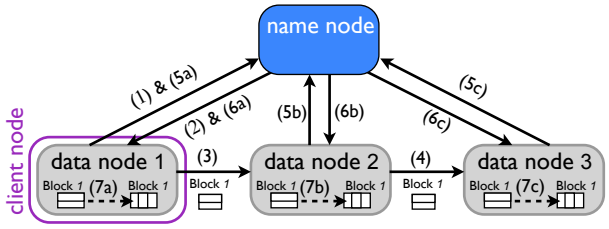


Figure 6: Process to upload a file to TROJAN HDFS

Layouts ids. As an example, the layout configuration file for TPC-H `Customers`, TPC-H `Lineitem`, SSB `LineOrder`, and SDSS `PhotoObj` can be as follows:

```
TPC-H_Customers: Row | Column | Customer_column_grouped
TPC-H_Lineitem: Row | Column | Lineitem_column_grouped
SSB_LineOrder: Row | Column | LineOrder_column_grouped
SDSS_PhotoObj: Row | Column | PhotoObj_column_grouped
```

The layout ids (e.g. Customer_column_grouped) in the above layout configuration file is mapped to actual attributes in a separate file. Users simply upload the layout configuration file into a predefined directory in TROJAN HDFS. At start up, the name node loads the layout configuration file into main memory. After this, the users can upload their data files into TROJAN HDFS exactly as in standard HDFS. Internally, TROJAN HDFS takes care of storing data block replicas in their respective Trojan Layouts, hiding all messy details from the users.

Figure 6 depicts the upload process with a replication factor of 3. For simplicity, we assume in Figure 6 that the data set to upload contains only a single data block. The idea is that as soon as all replicas of a data block are copied, the data nodes internally reorganize replicas according to their assigned Trojan Layout. In detail, the uploading process has the following steps: (1) the client node (e.g. data node 1) asks the name node to register a data block (e.g. data block 1); (2) the name node returns the set of data nodes to hold the three data block replicas (e.g. data nodes 1, 2, and 3); (3) after storing data block 1 locally, data node 1 sends a replica to data node 2; (4) data node 2 stores data block 1 locally and sends a replica to data node 3, which in turn also stores data block 1 locally; (5a)–(5c) each data node then informs the name node of the newly received data block 1; (6a)–(6c) the name node returns the Trojan Layout corresponding to the data block replica stored by each data node; (7a)–(7c) finally, each data node transforms data block 1 into its respective Trojan Layout. In case a user uploads a data set that is not in the layout configuration file, the name node asks the data nodes to keep the data layout unchanged (typically row layout).

4.3 Query Processing

To process an incoming MapReduce job, we first identify which attributes need to be read. We then use a cost model (Appendix A) to automatically pick the best Trojan Layout in TROJAN HDFS for the MapReduce job. Then, we schedule map tasks to those data nodes storing data block replicas in the best Trojan Layout (given by the cost model). We provide an *itemize* `UDF` [14] to the map tasks so that they can read only the referenced attributes and reconstruct tuples, all invisible to the users.

Algorithm 4 shows the `itemize.initialize` method for enabling a map task to transparently read referenced attributes from data blocks and automatically reconstruct tuples as expected by applications (MapReduce jobs). To do so, we first get the required attributes from the job configuration (Line 1). Then, we read the

Algorithm 4: Trojan Layout `itemize`.initialize UDF

```
   Input: FileSplit split, Configuration job
1  Set ReferencedAttributes = job.getRefAtts();
2  Global FileSplit split = split;
3  Header h = ReadHeader(split);
4  Set GroupedData, ReadGroups = ∅;
5  Global StringBuilder attributeOrder = new StringBuilder ();
6  foreach attribute in h.getAttributes() do
7      if ReferencedAttributes.contains(attribute) then
8          if !ReadGroups.contains(attribute.getStartOffset()) then
9              GroupedData.add(new Group
               (split.readfully(attribute.getStartOffset(),
               attribute.getEndOffset())));
10             ReadGroups = attribute.getStartOffset();
11         end
12         attributeOrder.append (attribute.getPosition() + ",");
13     end
14 end
```

header of a data block (Lines 2–3). The header information allows us to know the column groups that contains the referenced attributes (relevant column groups). We upload such relevant column groups into main memory (Line 6–10). Additionally, we keep track of the position of each referenced attribute so as to allow a map function to know how attributes are ordered in a tuple (Line 12). Now, to feed tuples to the `map` function, we simply iterate over each column group and check if a column group has more tuples. If so, we reconstruct the tuple from relevant column groups and pass it to the `map` function. Otherwise, we signal the end of tuples.

4.4 Scheduling Policies

By default, the Hadoop MapReduce scheduler tries to allocate map tasks to those data nodes having *any* replicas of the requested data block locally. However, with per-replica Trojan Layouts, scheduling map tasks to data nodes having different data block replicas may have quite different performance. For example, a MapReduce job requiring one attribute out of 16 would be a way faster to complete if the input data set is in column-layout. Therefore, to query per-replica Trojan Layouts, we always schedule map tasks to those nodes storing the best Trojan Layout (`Best-Layout` policy, for short). This policy is reasonable because in practice, if map tasks are slightly delayed [33], only 1% of map tasks need to fetch the best layout anyways. Still, in case of contention, we might end up delaying several map tasks with the `Best-Layout` scheduling policy. To avoid this delay, there could be two more scheduling policies to allocate these map tasks: (i) schedule map tasks to available nodes even if they do not store the best Trojan Layout; later, map tasks fetch data blocks with the best layout (`Fetch Best-Layout` policy), and (ii) schedule map tasks to those nodes storing the second best Trojan Layout; later, map tasks read the local data blocks (`2nd Best-Layout` policy). Both `Fetch Best-Layout` and `2nd Best-Layout` scheduling policies avoid delaying map tasks. However, `Fetch Best-Layout` policy now incurs networks costs to fetch the best layout whereas `2nd Best-Layout` policy affects the data access performance.

We experimentally compare these three scheduling policies in Section 5.5.

5. EXPERIMENTAL EVALUATION

We implemented our ideas on top of HDFS 0.20.3. We evaluate the performance of Trojan Layouts and compare it with Hadoop MapReduce 0.20.3 using Row Layout (Hadoop-Row) and PAX Layout (Hadoop-PAX). We ran our experiments with two main objectives in mind: (i) to show that the use of Trojan Layouts allows us to significantly improve data access performance, and (ii) to evaluate the effectiveness of our column grouping algorithm.

5.1 Testbed

We ran all our experiments on a physical 10-node cluster where each physical node runs five virtual nodes, using Xen virtualization, i.e., resulting in a total of 50 virtual nodes. Node virtualization is also used by Amazon to scale up its clusters. However, we showed that Amazon EC2 suffers from high variance in performance [30]. Therefore, running the experiments on our cluster allows us to get more stable results. Each physical node in our cluster has one 2.66 GHz Quad Core Xeon running 64-bit platform Linux openSuse 11.1 OS, 4x4 GB main memory, 6x750 GB SATA hard disks, and three Gigabit network cards. We set each virtual node to have a physical 750 GB hard disk and physical 3.2 GB main memory. The physical nodes are connected with a Cisco Catalyst 3750E-48PD, which uses two Gigabit Ethernet ports for each node in channel bonding mode. From now on, we refer to virtual nodes as *nodes* for clarity. We used Hadoop 0.20.3 running on Java 1.6 for all our experiments. We used Trojan HDFS to store input datasets, recall that Trojan HDFS is a variant of HDFS that supports different data layouts per-replica (see Section 4.2). We made the following three changes to the default HDFS settings: (i) we store data into Trojan HDFS using 256MB data blocks as in [14], (ii) we allow Hadoop to reuse the task JVM executor instead of restarting a new process per task, and (iii) we allow a node to concurrently run two map tasks and a single reduce task.

5.2 Datasets and Benchmarks

To better validate Trojan Layouts, we used three real-world datasets and benchmarks: TPC-H, Star Schema Benchmark (SSB), and the Sloan Digital Sky Survey[4] (SDSS).

TPC-H. We generated data for the `Customers` and `LineItem` tables using the TPC-H `DBGEN` data generator tool. We used a scale factor of 1,000 to generate 50 files of data, which results into a total of 23.74 GB for the `Customers` table and into a total of 759 GB for the `LineItem` table. Since TPC-H `Customers` table appears in only eight queries of the TPC-H benchmark, we consider only the first eight queries of all other tables as well.

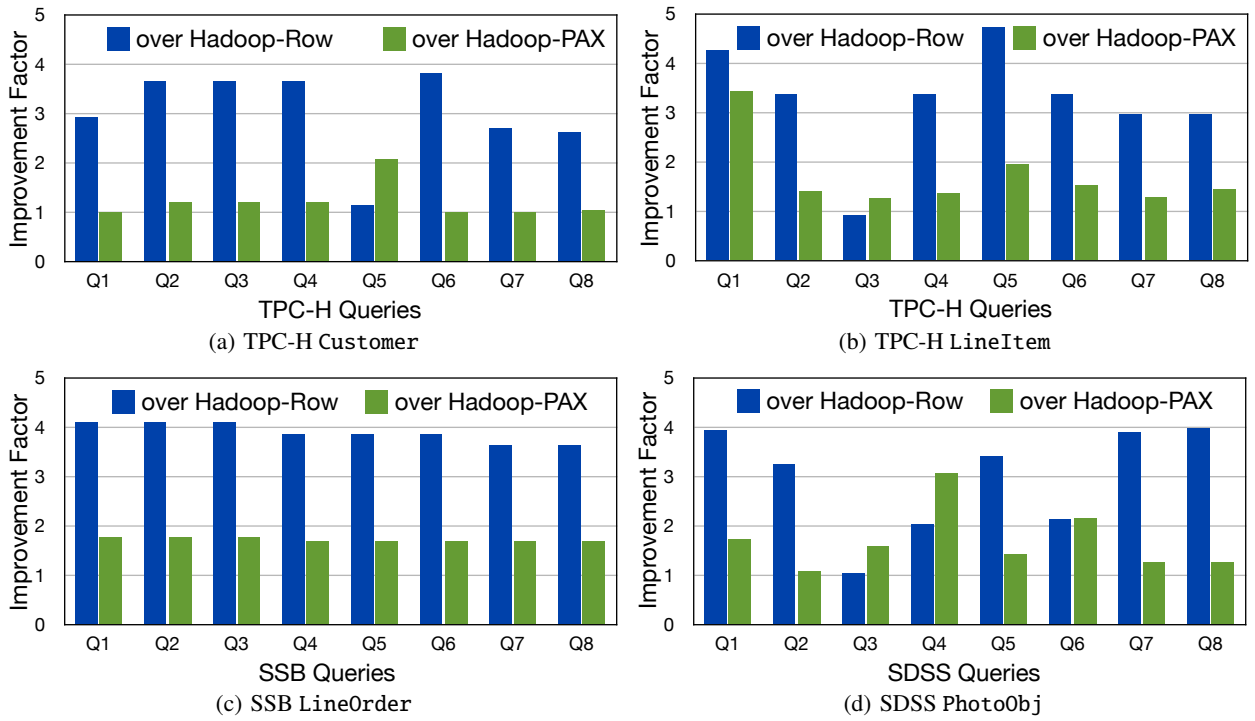
SSB. We generated data for `LineOrder` table using the SSB `DBGEN` tool. We used a scale factor of 1,000 to generate 50 files of data, which results into a total of 600 GB in total. We consider the first eight SSB queries, i.e. we use all three variants of the first two queries and the first two variants of the third query.

SDSS. We used ~50 GB of real-world data provided by SDSS for the `PhotoObj` table. As for TPC-H and SSB, we consider the first eight relevant SDSS queries. Notice that the `PhotoObj` table has 446 attributes in total. However, to be fair to Hadoop Row, for all three systems we consider only those 46 attributes which are referenced by our SDSS benchmark queries.

5.3 Methodology

We first evaluate how well Trojan Layouts allow MapReduce jobs to improve their performance. In particular, we evaluate how well our approach exploits different data replicas to fit a given workload. As this paper focuses on scan and projection operators of MapReduce jobs (see Figure 1), we implement map-phase-only MapReduce jobs for all our benchmark queries. The reason to do so is that Trojan Layouts improve the performance of MapReduce jobs by improving the way they read data from HDFS, which is done in the map phase of MapReduce jobs. Furthermore, we do not analyze the MapReduce job and assume that we know the data

[4]For further details on SDSS visit: http://www.sdss.org/



(a) TPC-H `Customer`

(b) TPC-H `LineItem`

(c) SSB `LineOrder`

(d) SDSS `PhotoObj`

Figure 7: Improvement of data access time when using Trojan Layouts over HADOOP-ROW and HADOOP-PAX.

access pattern, i.e. the attributes accessed by each query. Recent works in other aspects of MapReduce (shown in Figure 1) have described how to extract these data access patterns from MapReduce jobs [19, 7]. We run each benchmark three times, measure the time it takes to read the required data from disk — i.e. the elapsed time between the initialization and finalization of the `itemize` UDF — and report the improvement factor of our approach based on the average reading time of the trials.

5.4 Per-Replica Trojan Layout Performance

In this section, we evaluate the data access time improvement of Trojan Layouts over HADOOP-ROW and HADOOP-PAX. Let us first evaluate these three data layouts in terms of redundant attributes reads and attribute joins for tuple reconstruction. For this, we analyzed the query groupings and their Trojan Layouts (see Appendix B for layout details) and we observed that in all datasets at least two query groups fit perfectly to its corresponding Trojan Layout. Hence, per-replica Trojan Layouts significantly reduce redundant attribute access as well as tuple reconstruction overhead. Table 1 summarizes this observation.

	#Redundant Attributes Read	*#Joins in Tuple Reconstruction*
HADOOP-ROW	525	0
HADOOP-PAX	0	139
Trojan Layout	14	20

Table 1: Per-replica Trojan Layout analysis

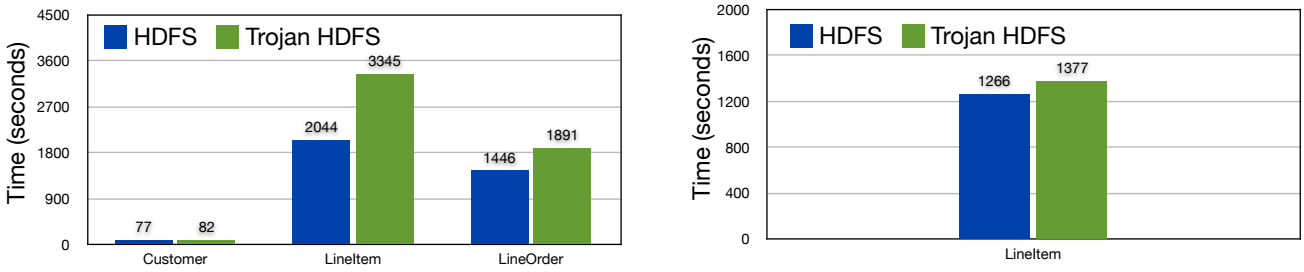
We observe that Trojan Layouts allow us to read ~37 times less redundant attributes than HADOOP-ROW and to perform ~7 times less attribute joins for reconstructing tuples than HADOOP-PAX. Thus, Trojan Layouts provide for a good trade-off between the number of redundant attributes and the number of joins in tuple reconstruction (green cells). This is in contrast to HADOOP-ROW and HADOOP-PAX, which are at the two extremes (red cells).

Figure 7 illustrates the improvement of data access time when using Trojan Layouts over HADOOP-ROW and HADOOP-PAX. We observe that for those queries referencing few attributes, e.g. Q_4 in `LineItem` and all queries in `LineOrder`, Trojan Layouts improve HADOOP-ROW up to factor of 4.8. Indeed, this is because HADOOP-ROW reads a large number of redundant attributes as shown in Table 1. In particular, we observe that HADOOP-ROW slightly outperforms Trojan Layouts only for Q_3 in `LineItem`. This is because all attributes are referenced and Trojan Layouts have an extra tuple reconstruction cost that HADOOP-ROW does not have. On the other side, we observe that for those queries referencing many attributes, e.g. Q_1 in `LineItem` and Q_4 in `PhotoObj`, Trojan Layouts outperform HADOOP-PAX up to a factor of 3.5. The reason is that tuple reconstruction cost in HADOOP-PAX increases as the number of referenced attributes increases as well. Trojan Layouts amortize tuple reconstruction cost by co-locating attributes in the same column groups. Further, the results show that Trojan Layouts never perform worse than HADOOP-PAX, having at least the same performance as HADOOP-PAX in the worst case (e.g. Q_6–Q_8 in `Customer`).

Overall, our experimental results show that Trojan Layouts significantly outperform HADOOP-ROW as well as HADOOP-PAX. Our experimental results also support the simulation results we presented in Figure 2.

5.5 Comparing Scheduling Policies

In the above experiments, we considered the `Best-Layout` scheduling policy (see Section 4.4), which always allocates map tasks to those nodes storing the best Trojan Layout for incoming map tasks. However, as discussed in Section 4.4, one could apply two other scheduling policies as well: the `Fetch Best-Layout` policy and the `2nd Best-Layout` policy. To understand which policy performs better, we measure their relative performance with



(a) Using 50 virtual nodes.

(b) Using 10 physical nodes.

Figure 8: Comparison of Data Loading Times in `Trojan` and standard HDFS.

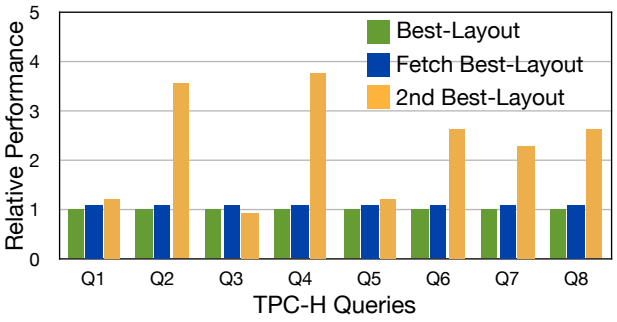


Figure 9: Worst-case relative data access performance when using different scheduling policies. We observe that the `2nd Best-Layout` policy significantly hurts performance for some queries, while the `Fetch Best-Layout` policy has an overhead of at most 9% over the `Best-Layout` policy. Therefore, in practice, one should try to use the best layout to perform queries even if data blocks has to be copied through the network.

respect to the `Best-Layout` policy over `TPC-H Lineitem` table. We compute the relative performance of a given scheduling policy as the ratio of the data access time of the given policy to the `Best-Layout` policy.

Figure 9 shows the results of these experiments. As expected, the `Best-Layout` policy performs better than the other two policies. However, we observe that the `Fetch Best-Layout` policy performs almost as well as the `Best-Layout` policy. This is not the case for the `2nd Best-Layout` policy, which is slower by a factor of up to ~3.8. This is because map tasks end up reading all attributes from disk in many cases. Thus, we can conclude that, when having data block replicas in different layouts, one should apply only the `Best-Layout` and `Fetch Best-Layout` policies.

5.6 Data Loading

Now we compare and analyze the data load performance of `Trojan` HDFS with standard HDFS. On a cluster of 50 virtual nodes, we consider the data load times of three data sets from our benchmarks: TPC-H `Customer`, TPC-H `Lineitem`, and SSB `LineOrder`. For each of these data sets, we load the data files on all data nodes in parallel, i.e. each of the fifty nodes loads ~470 MB of TPC-H `Customer` data (23.74 GB in total), ~15 GB of TPC-H `Lineitem` data (759 GB in total), and 12 GB of SSB `LineOrder` data (600 GB in total). We use the same command-line utility for both `Trojan` as well as standard HDFS.

Figure 8(a) illustrates the results of loading these three data sets into `Trojan` and standard HDFS. As expected, standard HDFS is faster than `Trojan HDFS` because it simply copies the data from local hard disks to the distributed file system. On the other hand, `Trojan` HDFS parses the data sets into binary representation and formats them into their Trojan Layout. However, from Figure 8(a), we see that the difference between the loading times of `Trojan` and standard HDFS becomes significantly high for larger tables, e.g. TPC-H `Lineitem` table. The reason for this overhead is that the `Trojan` HDFS is CPU-intensive due to data parsing and layouts transformation. However, because of node virtualization more than 60% of the CPU resources are already consumed. Furthermore, since each physical node of our cluster has a Quad-core processor (see Section 5.1), each virtual node gets only ~0.7 core. These two problems slow down the data loading in `Trojan` HDFS considerably. Standard HDFS, on the other hand, is I/O intensive and therefore does not get affected.

To actually verify our claims, we repeated the data loading experiments for `Lineitem` using only the 10 physical nodes, i.e., without any node virtualization. However, we still keep the amount of data per data node same. Figure 8(b) shows the loading times of `Trojan` and standard HDFS. We observe that `Trojan` HDFS now compares very well with standard HDFS. This is because the data nodes get much better CPU resources by not sharing the Quad-core processors anymore.

In summary, we can say that with appropriate cluster settings, the data load time overhead of `Trojan` HDFS is negligible. Furthermore, the one-time data load cost of `Trojan` HDFS pays off as recurring speed-ups over several MapReduce jobs.

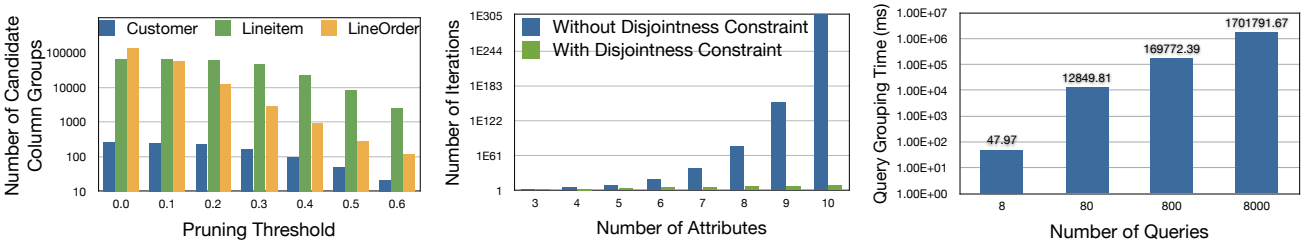
5.7 Comparison with HYRISE

In this section, we compare our column grouping algorithm with recently proposed HYRISE [16] layout selection algorithm. HYRISE proposes a cost-based divide and conquer technique for layout selection. It divides the set of candidate column groups using a k-way partitioner and then applies brute force search for the best layout per partition. Thereafter, it tries to merge column groups across partitions, before producing the final layout. This approach effectively improves upon the complexity of prior column grouping algorithms, e.g. [17]. However, it has two major problems: (i) there is little column grouping quality control, and (ii) query grouping, and hence per-replica layouts, is not possible.

In contrast, our interestingness-based column grouping algorithm takes the quality of column grouping into account. To illustrate this, we implemented HYRISE layout selection algorithm. Table 2 shows the redundant attributes accessed and tuple reconstruction joins in HYRISE and Trojan layouts.

	#Redundant Attributes Read	*#Joins in Tuple Reconstruction*
`HYRISE Layout`	2	64
`Trojan Layout`	14	20

Table 2: Quality Comparison of HYRISE and Trojan Layouts



(a) Pruning Effectiveness (b) Knapsack Iterations (c) Query Grouping Scalability

Figure 10: Performance of our column grouping algorithm.

We can see that even though HYRISE significantly reduces the redundant attributes accessed, it still incurs a very high tuple reconstruction cost. In contrast, Trojan Layout minimizes tuple reconstruction cost while allowing for additional (cheap) redundant reads. To verify our claim, we ran our benchmark queries over HYRISE layout. Indeed, our results showed over 14% improvement in total runtime of Trojan Layouts over HYRISE layouts on TPC-H `Lineitem`, TPC-H `Customer`, and SSB `LineOrder` tables. Similarly, Trojan Layouts have an improvement of 5.9% over HYRISE layouts on SDSS `PhotoObj` table. Lower improvement on `PhotoObj` table is due to the large number of attributes accessed by queries and the skewed query groups produced in our Trojan Layout (2 groups of 1 query each, and 1 group of 6 queries).

5.8 Grouping Algorithm Performance and Scalability

We now focus on the effectiveness of our algorithm to group attributes inside a data block.

Column Group Pruning. First of all, we show the effectiveness of our interestingness-based column group pruning. Figure 10(a) shows the pruning performance over TPC-H Customer, TPC-H `LineItem`, and SSB `LineOrder`. We observe that candidate column groups get pruned progressively with the pruning threshold.

Number of Iterations. Next, we show the effect of adding the disjointness constraint (Equation 2) to our knapsack formulation. Figure 10(b) compares the number of iterations with and without the disjointness constraint. Recall that the disjointness constraint prevented us from using dynamic programming algorithm. However, as we see from the figure, the disjointness constraint significantly reduces the number of iterations in our algorithm.

Query Workload Scalability. Finally, we show the scalability of our algorithm with query workload. Recall that to deal with large number of queries, we apply our grouping algorithm recursively, i.e. we first independently group sets of queries, then we independently group sets of query groups, and so on. Figure 10(c) shows the time taken to create query groups when scaling the number of queries. For instance, for $8,000$ queries, the time taken to group the queries is around 28 minutes. This is acceptable, given that grouping is an offline process. Thus, our algorithm scales well with query workload size.

6. RELATED WORK

Column Layouts in Traditional Systems. As an alternative to traditional n-ary storage model, Decomposition Storage Model (DSM) [12] was the earliest approach to store data in a column-oriented layout, i.e. all values of an attribute stored together. Later, researchers proposed PAX [6], a page-level column layout, to improve cache performance. Finally, Column-Stores [1] reinvented DSM to significantly improve upon storage requirements as well as query processing. However, all these approaches were designed for single-node data processing systems and thus do not care about replication. In a distributed setting, Fractured Mirrors [28] keeps two replicas of data: one in row and the other in column layout. However, it has a fixed number of replicas (two) as well as layouts (row and column). Trojan layouts, on the other hand, can work with any number of replicas and can even create hybrid layouts. Still, in special cases, Trojan layouts can mimic Fractured Mirrors.

Column Layouts in MapReduce. Row layouts in MapReduce could incur significant overheads over large datasets. To deal with this, recent works such as Cheetah [11] and ES2 [8] propose to use block-level PAX layouts in MapReduce. Similarly, a more recent work [15] creates, for each horizontal range, a different physical file per attribute. However, all these works create identical layouts for different replicas of a data block and do not consider column grouping. In contrast, Trojan Layouts can create a different layout per data block replica and also consider column grouping.

Column Grouped Layouts. Given a query workload, finding the optimal column grouping (or vertical partitioning) is a NP-hard problem [29]. Thus, most of the works on vertical partitioning, including the initial approach [23], Data Morphing [17], and HYRISE [16] focus on heuristics to improve the runtime complexity but do not consider the quality of column grouping. [5] considers the interestingness (CG-Cost) of candidate column groups and prunes the ones below a certain threshold. However, CG-Cost has several problems (discussed in Section 3.1). Our interestingness measure significantly improves upon CG-Cost, thus enabling our column grouping algorithm to produce better quality results. Furthermore, previous column grouping algorithms produce a single best layout for the entire workload. However, in a parallel data processing system having inherent data replication, different replicas could be mapped to different layouts. Our column grouping algorithm makes this possible by producing a set of column-grouped layouts, each of which are best suited for a different subset of the workload. With this, we can later route an incoming query to a more specialized layout.

DBMS Stores with MapReduce. HadoopDB [4] replaces the HDFS storage in MapReduce with a database. Thus, the data is now in the DBMS data layout (row layout for row-oriented DBMS, column layout for column-oriented DBMS). However, this involves severe changes to the Hadoop execution framework. Similarly, another approache analyzes the map functions and translate them into SQL queries to be run on a database [19]. These approaches are orthogonal to our *Trojan* philosophy: affect Hadoop from inside in a non-invasive manner by injecting our technology at the right places through UDFs only [14].

7. CONCLUSION

MapReduce suffers from very slow execution times in some analytical tasks compared to DBMSs. One of the reasons for this is that

MapReduce processes input data blocks in a strictly row-oriented fashion, which leads to full scans of the input data [26].

In this paper we proposed Trojan Layouts, a new data layout that organizes data inside data blocks according to the incoming workload. We followed the PAX principle in that we did not change the outside view of data. However, we considerably depart from PAX as we: (i) might co-locate attributes together according to query workloads, (ii) may use different Trojan Layouts for different data block replicas, and (iii) may, in a special case, mimic fractured mirrors: having the best from both PAX and Row Layouts. We implemented our algorithms on top of HDFS 0.20.3. A salient feature of using per-replica Trojan Layouts is that we can schedule incoming jobs to data block replicas having the best Trojan Layout.

We experimentally evaluated Trojan Layouts using three real-world benchmarks: TPC-H, SSB, and SDSS, and compared its effectiveness against Hadoop using Row (Hadoop-Row) and PAX (Hadoop-PAX) layouts. The results demonstrated that our approach significantly outperforms both Hadoop-Row and Hadoop-PAX in all three benchmarks: up to a factor of 4.8 for Hadoop-Row and up to a factor of 3.5 for Hadoop-PAX. Figure 11 illustrates how the experimental runtimes of queries Q1 to Q8 varies with the number of referenced attributes for TPC-H `Customer` table. In particular,

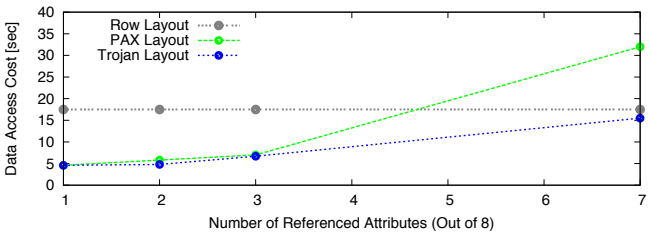


Figure 11: Simulation Validation for TPC-H `Customer`

the results showed that the performance of Hadoop-PAX decreases quickly as the number of referenced attributes increases. This is not the case for Trojan Layouts. In other words, our experimental results support the simulation results we presented in the introduction of this paper (see Figure 2).

As future work, we plan to adapt Trojan Layouts to changes in the workload. Several strategies, such as piggy backing into ongoing MapReduce jobs, can be employed and need to be investigated in more detail.

8. REFERENCES

[1] D. Abadi, P. Boncz, and S. Harizopoulos. Column-Oriented Database Systems. *PVDLB*, 2(2), 2009.

[2] D. Abadi et al. Materialization Strategies in a Column-Oriented DBMS. In *ICDE*, 2007.

[3] D. Abadi, S. Madden, and N. Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In *SIGMOD*, 2008.

[4] A. Abouzeid et al. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1), 2009.

[5] S. Agrawal et al. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *SIGMOD*, 2004.

[6] A. Ailamaki et al. Weaving Relations for Cache Performance. In *VLDB*, 2001.

[7] M. J. Cafarella and C. Ré. Manimal: Relational Optimization for Data-Intensive Programs. In *WebDB*, 2010.

[8] Y. Cao et al. A Cloud Data Storage System for Supporting Both OLTP and OLAP. In *ICDE*, 2011.

[9] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.

[10] S. Chaudhuri. Self-Tuning Database Systems: A Decade of Progress (Ten Year Best paper Award). In *VLDB*, 2007.

[11] S. Chen. Cheetah: A High Performance, Custom Data Warehouse on Top of MapReduce. *PVLDB*, 3(2), 2010.

[12] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *SIGMOD*, 1985.

[13] J. Dean and S. Ghemawat. MapReduce: A Flexible Data Processing Tool. *CACM*, 53(1):72–77, 2010.

[14] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *PVLDB*, 3(1), 2010.

[15] A. Floratou et al. Column-Oriented Storage Techniques for MapReduce. *PVLDB*, 4(7), 2011.

[16] M. Grund et al. HYRISE - A Main Memory Hybrid Storage Engine. *PVLDB*, 4(2), 2010.

[17] R. A. Hankins and J. M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *VLDB*, 2003.

[18] R. Ikeda and J. Widom;. Provenance for Generalized Map and Reduce Workflows. In *CIDR*, 2011.

[19] M.-Y. Iu and W. Zwaenepoel. HadoopToSQL: A MapReduce Query Optimizer. In *EuroSys*, 2010.

[20] W. Lang and J. M. Patel. Energy Management for MapReduce Clusters. *PVLDB*, 3(1), 2010.

[21] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[22] K. Morton, M. Balazinska, and D. Grossman. ParaTimer: A Progress Indicator for MapReduce DAGs. In *SIGMOD*, 2010.

[23] S. Navathe et al. Vertical Partitioning Algorithms for Database Design. *ACM TODS*, 9(4):680–710, 1984.

[24] T. Nykiel et al. MRShare: Sharing Across Multiple Queries in MapReduce. *PVLDB*, 3(1), 2010.

[25] C. Olston et al. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD Conference*, 2008.

[26] A. Pavlo et al. A Comparison of Approaches to Large-Scale Data Analysis. In *SIGMOD*, 2009.

[27] J.-A. Quiané-Ruiz et al. RAFTing MapReduce: Fast Recovery on the Raft. In *ICDE*, 2011.

[28] R. Ramamurthy, D. J. DeWitt, and Q. Su. A Case for Fractured Mirrors. In *VLDB*, 2002.

[29] D. Sacca and G. Wiederhold. Database Partitioning in a Cluster of Processors. *ACM TODS*, 10(1):29–56, 1985.

[30] J. Schad, J. Dittrich, and J. Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *PVLDB*, 3(1), 2010.

[31] M. Stonebraker et al. C-Store: A Column-Oriented DBMS. In *VLDB*, 2005.

[32] A. Thusoo et al. Data Warehousing and Analytics Infrastructure at Facebook. In *SIGMOD*, 2010.

[33] M. Zaharia et al. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys*, 2010.

APPENDIX

A. COST MODEL

In this section, we describe our cost model as well as the parameters we used for our simulations in Figure 2. We model the costs for full table scan access over four different layouts: (i) Row Layout, (ii) Column Layout, (iii) PAX Layout, and (iv) Optimal Layout (which contains, for each query, perfect column groupings within a data block). Table 3 shows the cost model for these layouts and Table 4 lists the symbols used in our cost model.

Our cost model considers random and sequential I/Os to read data from HDFS, network costs to transfer data blocks across data nodes, and even the scheduling decisions made by MapReduce scheduler. For network costs, we compute the probability of not finding any local data block copy and estimate the costs of transferring data from another node. Note that the Row and PAX layouts have same network transfer costs. On the other hand Column and PAX layouts have same random and sequential I/O costs, as

Symbol	Meaning	Model
w	# map phases (waves)	$\left\lceil \frac{N \cdot B}{S \cdot m \cdot n} \right\rceil$
$C_{\text{tr}}^{\text{row}}(S)$	transfer cost for row layout	$(1 - p_{1\text{r}}) \frac{S}{\text{BW}_{\text{net}}}$
$C_{\text{rand}}^{\text{row}}(S)$	rand I/O cost for row layout	$C_{\text{rand}} \cdot \frac{S}{b}$
$C_{\text{seq}}^{\text{row}}(S)$	seq I/O cost for row layout	$\frac{S}{\text{BW}_{\text{disk}}}$
$C_{\text{tr}}^{\text{opt}}(S)$	transfer cost for optimal layout	$(1 - p_{1\text{r}}) \frac{S}{\text{BW}_{\text{net}}}$
$C_{\text{rand}}^{\text{opt}}(S)$	rand I/O cost for optimal layout	$C_{\text{rand}} \cdot \frac{S \cdot \lvert A' \rvert}{b \cdot \lvert A \rvert}$
$C_{\text{seq}}^{\text{opt}}(S)$	seq I/O cost for optimal layout	$\frac{S \cdot \lvert A' \rvert}{\text{BW}_{\text{disk}} \cdot \lvert A \rvert}$
$C_{\text{tr}}^{\text{pax}}(S)$	transfer cost for PAX layout	$(1 - p_{1\text{r}}) \frac{S}{\text{BW}_{\text{net}}}$
$C_{\text{rand}}^{\text{pax}}(S)$	rand I/O cost for PAX layout	$C_{\text{rand}} \cdot \left\lceil \frac{S \cdot \lvert A' \rvert}{b \cdot \lvert A \rvert} \right\rceil \cdot \lvert A' \rvert$
$C_{\text{seq}}^{\text{pax}}(S)$	seq I/O cost for PAX layout	$\frac{S \cdot \lvert A' \rvert}{\text{BW}_{\text{disk}} \cdot \lvert A \rvert}$
$C_{\text{tr}}^{\text{col}}(S)$	transfer cost for column layout	$\left[1 - p_{1\text{r}} + \left(1 - \frac{R}{n}\right) \cdot (\lvert A' \rvert - 1)\right] \frac{S}{\text{BW}_{\text{net}}}$
$C_{\text{rand}}^{\text{col}}(S)$	rand I/O cost for column layout	$C_{\text{rand}} \cdot \left\lceil \frac{S \cdot \lvert A' \rvert}{b \cdot \lvert A \rvert} \right\rceil \cdot \lvert A' \rvert$
$C_{\text{seq}}^{\text{col}}(S)$	seq I/O cost for column layout	$\frac{S \cdot \lvert A' \rvert}{\text{BW}_{\text{disk}} \cdot \lvert A \rvert}$
$C_{\text{scan}}^{\text{row}}$	scan cost for row layout	$\left(C_{\text{tr}}^{\text{row}}(S) + C_{\text{rand}}^{\text{row}}(S) + C_{\text{seq}}^{\text{row}}(S) + C_{\text{init}}^{m}\right) \cdot w$
$C_{\text{scan}}^{\text{opt}}$	scan cost for optimal layout	$\left(C_{\text{tr}}^{\text{opt}}(S) + C_{\text{rand}}^{\text{opt}}(S) + C_{\text{seq}}^{\text{opt}}(S) + C_{\text{init}}^{m}\right) \cdot w$
$C_{\text{scan}}^{\text{pax}}$	scan cost for PAX layout	$\left(C_{\text{tr}}^{\text{pax}}(S) + C_{\text{rand}}^{\text{pax}}(S) + C_{\text{seq}}^{\text{pax}}(S) + C_{\text{init}}^{m}\right) \cdot w$
$C_{\text{scan}}^{\text{col}}$	scan cost for column layout	$\left(C_{\text{tr}}^{\text{col}}(S) + C_{\text{rand}}^{\text{col}}(S) + C_{\text{seq}}^{\text{col}}(S) + C_{\text{init}}^{m}\right) \cdot w$

Table 3: Full table scan access cost model for different layouts

Symbol	Meaning	Unit	Default Value
N	number of blocks		400
B	block size	bytes	256 MB
S	split size	bytes	256 MB
R	replication factor		3
n	number of nodes		50
m	number of concurrent map tasks	2	
C_{init}^{m}	map initialization cost	seconds	0.1 sec
C_{rand}	random seek cost	seconds	0.005 sec
BW_{disk}	disk bandwidth	bytes/s	100 MB/s
BW_{net}	network bandwidth	bits/s	1 GBits/s
b	buffer size	bytes	512 KB
$p_{1\text{r}}$	probability of first replica being local		0.97
A	attribute set		{1,..,30}
A'	referenced attribute set		{1},{1,2}..

Table 4: Cost Model Symbols

Hadoop performs a buffered read anyways. However, these two layouts have different network costs.

Table 4 also shows the default parameter values which we used in our simulation. We assume block size to be equal to split size. We assume 400 blocks in total, which is equivalent to 1TB of data size, over a cluster of 50 nodes. We experimentally determined a map initialization time of 0.1 sec and consider random I/O cost to be 0.005 sec. Our disk and network bandwidths are 100 MB/s and 1 GBits/s respectively. Based on the results presented in [33], we assume the probability of finding the first replica locally to be 0.97. Finally, we consider a dataset with 30 attributes.

Now let us see the estimated cost of different layouts on different benchmarks. Figure 12 shows the runtime simulation of data access over Row, Column, PAX, and Trojan (perfectly column-grouped) data layouts on TPC-H Customer, TPC-H Lineitem, SSB LineOrder, and SDSS PhotoObj datasets. We can see that PAX and column-grouped PAX outperforms the traditional Row and Column layouts by up to 10 orders of magnitude. Furthermore, Trojan layout outperforms PAX by a factor of up to 1.4. Note that we model just the I/O costs here. In practice, PAX layouts incur significant CPU costs for tuple reconstruction as well. Thus, we expect the actual improvements of Trojan layout to be much higher in practice.

B. LAYOUT DETAILS

In this section, we show and discuss the Trojan Layouts that we obtained from our algorithms when running our experiments in Section 5. Since we create Trojan data layouts per replica of a given data block, let us first look at the query groupings generated for our experimental datasets. Recall that we use the same algorithm for generating query groups as we use for generating column groups i.e. we simply invert the attribute usage matrix. Figure 5 shows the query groups for the first relevant eight queries on TPC-H `Customer`, TPC-H `Lineitem`, SSB `LineOrder`, and SDSS `PhotoObj` datasets. Here, we assume three replicas per data block and generate three query groups, one for each replica.

Dataset	Replica 1	Replica 2	Replica 3
TPC-H `Customer`	Q_2, Q_3, Q_4	Q_5	Q_1, Q_6, Q_7, Q_8
TPC-H `Lineitem`	Q_1	Q_5	Q_2, Q_3, Q_4, Q_6, Q_7, Q_8
SSB `LineOrder`	$Q1$, $Q2$, $Q3$	$Q4$, $Q5$, $Q6$	Q_7, Q_8
SDSS `PhotoObj`	Q_4	Q_6	Q_1, Q_2, Q_3, Q_5, Q_7, Q_8

Table 5: Query Grouping

Note that two query groups in TPC-H `Customer`, TPC-H `Lineitem`, and SDSS `PhotoObj` datasets as well as all three query groups in SSB `LineOrder` dataset match perfectly with the attribute access pattern. This means there is no redundant attribute accessed and there are no joins in tuple reconstruction. We map each query group to a replica and then compute the column grouping for each replica separately.

Column Groups	Replica 1	Replica 2	Replica 3
CG_1	1,2,4,5,6,7	6	2,3,7
CG_2	0,3	0,1,2,3,4,5,7	0
CG_3			1
CG_4			4,5
CG_5			6

Table 6: TPC-H `Customer` Column Groups. Green replicas have perfect Trojan Layouts for the queries routed to them.

Column Groups	Replica 1	Replica 2	Replica 3
CG_1	0,1,2,3,11, 12,13,14,15	0,1,2,3,7,8,9 11,12,13,14,15	0,2,5,6,10
CG_2	4,5,6,7,8,9,10	4,5,6,10	1,4
CG_3			8,9,11
CG_4			14,15
CG_5			7,12
CG_6			3,13

Table 7: TPC-H `Lineitem` Column Groups. Green replicas have perfect Trojan Layouts for the queries routed to them.

Column Groups	Replica 1	Replica 2	Replica 3
CG_1	0,1,2,3,4,6 7,10,12,13 14,15,16	0,1,2,6,7 8,9,10,11 13,14,15,16	0,1,3,6,7 8,9,10,11 13,14,15,16
CG_2	5,8,9,11	3,4,5,12	2,4,5,12

Table 8: SSB `LineOrder` Column Groups. Green replicas have perfect Trojan Layouts for the queries routed to them.

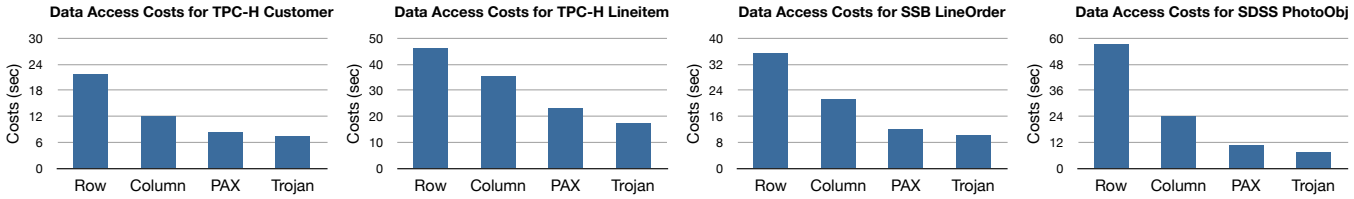


Figure 12: Estimated Running Time for Various Benchmarks

Column Groups	Replica 1	Replica 2	Replica 3
CG_1	2,3,4,5,8,9 10,11,12,13,14 15,16,17,18,19 20,21,22,28,29 39,40,41,42,43 44,45	1,2,3,4,5,6,7,13 14,15,16,17,18 19,20,21,22,23 24,25,26,27,28 29,31,32,33,34 35,36,38,39,45	13,14,15,16,24 25,26,27,30,31 32,33,34,35,36 38,40,41,42,43 44,45
CG_2	0,1,6,7,23,24 25,26,27,30,31 32,33,34,35,36 37,38	0,8,9,10,11,12 30,37,40,41,42 43,44	0,1,2,3
CG_3			4,5
CG_4			37,39
CG_5			11,19
CG_6			7,28
CG_7			8,23
CG_8			12,18
CG_9			10,20
CG_{10}			17,21
CG_{11}			6,9
CG_{12}			22,29

Table 9: SDSS `PhotoObj` Column Groups. Green replicas have perfect Trojan Layouts for the queries routed to them.

Tables 6, 7, 8, and 9 show the column groupings for the three data block replicas of TPC-H `Customer`, TPC-H `Lineitem`, SSB `LineOrder`, and SDS `PhotoObj` tables respectively. In these tables, we represent the attributes as integer (starting from 0) attribute IDs in the same sequence as they appear in their benchmark datasets. For instance, in Table 6, attribute IDs 0 to 7 denote the eight attributes of TPC-H `Customer` table. Note that each replica may have a different number of column groups, e.g. SDSS `PhotoObj` has two column groups each in the first two replicas whereas it has twelve column groups in the last replica (see Table 9). However, the union of columns groups in each of the replicas contains all attributes present in the dataset. It is worth to notice that at least two replicas (shown as green in Tables 6, 7, 8, and 9) of each dataset perfectly fit the queries routed to them i.e. the queries do not access any redundant attributes nor need any joins for tuple reconstruction.

Finally, observe that increasing the number of replicas allows us to create more variety of column groupings and thus hence fit a heterogenous query workload better. In the extreme case, we could maintain one replica, with perfect column grouping, for each query in the workload. However, the downside is that such an arrangement incurs exorbitant storage costs. Nevertheless, the beauty of Trojan Layouts is that it exploits the default replication in parallel data processing systems without touching the distributed data storage configurations.

C. BENCHMARKS QUERIES

In this section we enumerate the workload queries used in our experiments (see Section 5). For each of the four tables — TPC-H `Customer`, TPC-H `Linitem`, SSB `LineOrder`, and SDSS `PhotoObj` — we pick those first eight queries, in their respective benchmarks, which touch at least one attribute from them. The reason for doing this was that only eight TPC-H queries access any of the attributes in Customer table. Hence, in order to be fair and have equal-sized workload for all datasets, we picked just the first eight queries over the four datasets. Tables 10, 11, 12,and 13 below list the queries which we use over the three layouts — Row Layout, PAX Layout, and Trojan Layout — in our experiments.

Query Number	Query	Referenced Attributes
Q_1	TPC-H Query 3	0,6
Q_2	TPC-H Query 5	0,3
Q_3	TPC-H Query 7	0,3
Q_4	TPC-H Query 8	0,3
Q_5	TPC-H Query 10	0,1,2,3,4,5,7
Q_6	TPC-H Query 13	0
Q_7	TPC-H Query 18	0,1
Q_8	TPC-H Query 22	0,4,5

Table 10: TPC-H `Customers` Queries (#total attributes = 8)

Query Number	Query	Referenced Attributes
Q_1	TPC-H Query 1	4,5,6,7,8,9,10
Q_2	TPC-H Query 3	0,5,6,10
Q_3	TPC-H Query 4	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
Q_4	TPC-H Query 5	0,2,5,6
Q_5	TPC-H Query 6	4,5,6,10
Q_6	TPC-H Query 7	0,2,5,6,10
Q_7	TPC-H Query 8	0,1,2,5,6
Q_8	TPC-H Query 9	0,1,2,4,5,6

Table 11: TPC-H `Lineitem` Queries (#total attributes = 16)

Query Number	Query	Referenced Attributes
Q_1	SSB Query 1.1	5,8,9,11
Q_2	SSB Query 1.2	5,8,9,11
Q_3	SSB Query 1.3	5,8,9,11
Q_4	SSB Query 2.1	3,4,5,12
Q_5	SSB Query 2.2	3,4,5,12
Q_6	SSB Query 2.3	3,4,5,12
Q_7	SSB Query 3.1	2,4,5,12
Q_8	SSB Query 3.2	2,4,5,12

Table 12: SSB `LineOrder` Queries (#total attributes = 17)

Query Number	Query	Referenced Attributes
Q_1	Basic SELECT-FROM-WHERE	0,1,2,3
Q_2	Moving Asteroids	0,4,5
Q_3	Using three tables	0,2,3,6,7,8,9,10,11,12,17 18,19,20,21,22,23,28,29
Q_4	Selected neighbors in run	0,1,6,7,23,24,25,26,27,30 31,32,33,34,35,36,37,38
Q_5	Gridded galaxy counts	2,3,37,39
Q_6	Stars multiply measured	0,8,9,10,11,12,30,37 40,41,42,43,44
Q_7	Spatial Queries with HTM functions	0,2,3
Q_8	Checking if objects are in SDSS footprint	0,2,3

Table 13: SDSS `PhotoObj` Queries (#total attributes = 46)