ATUN-HL: Auto Tuning of Hybrid Layouts using Workload and Data Characteristics

Rana Faisal Munir[1,2], Alberto Abelló[1], Oscar Romero[1], Maik Thiele[2], and Wolfgang Lehner[2]

[1] Universitat Politècnica de Catalunya (UPC), Barcelona, Spain
{fmunir,aabello,oromero}@essi.upc.edu
[2] Technische Universität Dresden (TUD), Dresden, Germany
{maik.thiele,wolfgang.lehner}@tu-dresden.de

Abstract. Ad-hoc analysis implies processing data in near real-time. Thus, raw data (i.e., neither normalized nor transformed) is typically dumped into a distributed engine, where it is generally stored into a hybrid layout. Hybrid layouts divide data into horizontal partitions and inside each partition, data are stored vertically. They keep statistics for each horizontal partition and also support encoding (i.e., dictionary) and compression to reduce the size of the data. Their built-in support for many ad-hoc operations (i.e., selection, projection, aggregation, etc.) makes hybrid layouts the best choice for most operations.
Horizontal partition and dictionary sizes of hybrid layouts are configurable and can directly impact the performance of analytical queries. Hence, their default configuration cannot be expected to be optimal for all scenarios. In this paper, we present ATUN-HL (Auto TUNing Hybrid Layouts), which based on a cost model and given the workload and the characteristics of data, finds the best values for these parameters. We prototyped ATUN-HL for Apache Parquet, which is an open source implementation of hybrid layouts in Hadoop Distributed File System, to show its effectiveness. Our experimental evaluation shows that ATUN-HL provides on average 85% of all the potential performance improvement, and 1.2x average speedup against default configuration.

Keywords: Big data, Hybrid storage layouts, Auto tuning, Parquet

1 Introduction

Data analysis plays a decisive role in todays data-driven organizations, which increasingly produce and store large volumes of data in the order of petabytes to zettabytes [16]. The storage and processing of such data has imposed a shift in the hardware, from single machines to large scale distributed systems. Apache Hadoop[3] is a pioneer large-scale distributed system and consists of a storage layer, namely Hadoop Distributed File System (HDFS)[4], and a processing layer, namely MapReduce[6]. The former allows to keep data in raw format without any

[3] https://hadoop.apache.org
[4] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

normalization or pre-processing. The latter allows data-intensive flows (DIFs) to process raw data such that they are ready for the analysis.

Hadoop and many modern in-memory processing engines (i.e., Apache Spark[5]) provide high-level languages (i.e., Apache Pig, Hive, and SparkSQL) that facilitate writing DIFs for processing raw data (e.g., removing dirty data, integrating multiple data sources) stored in HDFS. Typically, the processed data is stored as a very wide table for analytical queries, because of its advantages over normalized tables [4, 12]. Hybrid layouts are de-facto preferred options for storing such wide tables, due to their built-in support for many basic operations (i.e., selection, projection, aggregation, etc.) allowing ad-hoc analysis, without the need of moving the data to other storage (i.e., relational, document store, etc.).

There are several available hybrid layout implementations, such as: Optimized Record Columnar (ORC)[6], Parquet[7] and CarbonData[8]. All of them follow the same physical structure. Data is stored into multiple horizontal partitions, known as stripes in ORC, row groups (RGs) in Parquet and blocklet in CarbonData, and each horizontal partition stores its data column-wise. Hybrid layouts also store min-max statistics [13] for each horizontal partition to help in filtering (i.e., partitions that do not match predicates of a query are skipped). In addition, they support dictionary encoding to encode repetitive values, that can also be used for further filtering partitions.

	Small Partition	Large Partition
Parallelism	+	-
Task overhead	-	+
Filtering	+	-
Metadata size	-	+
Dictionary encoding	-	+
Memory buffering	+	-
Load balancing	+	-

Table 1: Effect of horizontal partition size

Despite having default values, the sizes of horizontal partitions and dictionary are configurable, depending on the type of workload. Thus, their values should be decided based on the data characteristics and usage. For instance, it is recommended to have a small size of horizontal partition for low selectivity queries and, a large size for high selectivity queries. However, it is not straight-forward to find an optimal size for all the queries, because this depends on their concrete selectivity and the type of data they access, therefore the problem becomes challenging. Moreover, the size of horizontal partitions can also effect different execution settings, which is shown in Table 1. It can be seen that small partitions positively impacts parallelism (by increasing the number of parallel tasks), filtering (by skipping unmatched partitions using statistics), memory buffering (they require less memory to buffer the data before flushing to the disk), and load balancing (by better distributing the loads among multiple machines). Whereas,

[5]https://spark.apache.org
[6]https://orc.apache.org
[7]https://parquet.apache.org
[8]https://carbondata.apache.org

large horizontal partitions help positively to reduce task overhead (by reading less metadata and reducing Java garbage collector overhead), metadata size (by storing less statistics), and also helps in performing better encoding (by encoding large number of repetitive values). In this paper, we aim at improving filtering, metadata size and dictionary encoding by choosing the optimal partition size.

Similarly, the characteristics of data require different dictionary sizes to handle different attribute lengths and number of distinct values. The dictionary is not only important for compression, but it can also be used to filter partitions. Specifically, when data is unsorted and it is not possible to filter partitions simply using min-max statistics, as we will show in Section 5.3.

In this paper, we present our approach, namely ATUN-HL, which helps to find best values for the aforementioned parameters using a cost model, which estimates the optimal values for the size of the horizontal partition and the dictionary, based on the given workload and data characteristics. Moreover, it should also be noted that the chunk size of HDFS is always greater than or equal to the horizontal partition size. Hence, it should be configured accordingly. We instantiated ATUN-HL for Parquet, to show its applicability in real scenarios and conducted an extensive evaluation on TPC-H[9] to show that ATUN-HL can significantly improve the query response times over Parquet with default configuration.

The main contributions of this work can be summarized as follows:

- We extend the cost model for hybrid layouts presented in [14].
- We propose ATUN-HL, a framework to optimize hybrid layouts.
- We prototype ATUN-HL on Parquet to show its benefits.
- We report the results of our extensive evaluation with TPC-H benchmark.

The remaining paper is organized as follows. In Section 2, we discuss the related work. In Sections 3 and 4, we discuss the cost model and our approach in detail. In Section 5, we show our experimental results. Finally, in Section 6, we conclude the paper.

2 Related Work

In [7], an indexing technique is proposed for hybrid layouts. The indexing information is stored as metadata per RG and data node, which in turn enables filtering RGs in selective queries. However, this approach uses default RG size, which as previously argued does not always perform well.

In [17, 18], different partitioning approaches are presented, which help in selective queries. In [17], data is divided into multiple horizontal partitions and in each partition, data is stored row-wise, rather than column-wise. It also stores extra meta information for each partition, which is computed based on the predicates of queries. Predicates are used as features, where a bit is stored for each tuple matching a feature. This eventually gives a feature-vector for every tuple, which is then used for filtering partitions. A similar vector is also used in [18], however this time it utilizes hybrid layouts with column grouping, instead of

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

fixed row layouts. The latter helps for both selection and projection queries. Yet, these techniques fall short when it comes to tuning the configurable parameters and they only provide new strategies of partitioning.

In [4], a column reordering technique is proposed to reduce the disk seek cost for hybrid layouts by storing together the columns which are accessed by the same queries. In addition, this approach sometimes duplicates columns to store them in a contiguous way. It helps to reduce the disk seek cost and overall improves the query execution time. However, it still does not try to find the optimal configuration values of hybrid layouts based on the running workload.

There are other works [2, 3, 11, 15], which try to use different layouts based on the workload. The goal is always to store the data in the most appropriate one for the given workload. Yet again, they do not optimize the layouts.

There are still few research works [9, 10] available on tuning big data analytical platforms (such as Hadoop). They focus on finding the optimal values for each configuration parameter available in a big data analytical system. Nevertheless, they target overall systems rather then individual layouts. These techniques can be used as complementary to our approach.

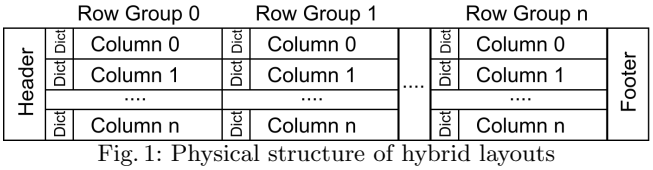

Fig. 1: Physical structure of hybrid layouts

3 Cost Model

In this section, we extend a cost model of our previous work [14]. Specifically, we refine the selection cost model based on the use of min-max statistics and dictionary encoding. Further, we extend our cost model to estimate the dictionary size for hybrid layouts.

First, we present the physical structure of hybrid layouts, which helps to build the cost model. Based on that, we estimate the cost of selections and the size of the dictionary. The former helps to find the optimal RG size. Our cost model considers two scenarios to estimate the selection cost, which are as follows: filtering using min-max statistics and using the dictionary. Likewise, it considers two types of dictionaries, i.e., global and local.

As shown in Figure 1, the data is divided into RGs (i.e., horizontal partitions), and inside each RG, it is stored column-wise. Further, if dictionary encoding is possible, first dictionaries are stored per column and afterwards the corresponding encoded data. If dictionary encoding is not possible, then the data values are stored contiguously without any encoding. Moreover, hybrid layouts also store metadata (e.g., min-max statistics) for each RG inside either the header or footer section. Thus, the size of hybrid layouts depends on the size of the actual data and metadata.

[10]Extra 4 bytes are considered for variable length columns

Variable	Description
	System Constants
p	Probability of accessed replica being local
$Chunk_{Size}$	Disk assignment unit size in HDFS
BW_{Disk}	Disk bandwidth
BW_{Net}	Network bandwidth
$Time_{Seek}$	Disk seek time
$Time_{Disk}$	$Chunk_{Size}/BW_{Disk}$
$Time_{Net}$	$Chunk_{Size}/BW_{Net}$
	Data Statistics
$\|T\|$	Number of rows in a table
$ColValue_{Size}$[10]	Average size of a column value
$\#Cols$	Total columns of T
$\|C\|$	Distinct values of a column
$\|D\|$	Number of values in the dictionary
$Sorted_{Col}$	True for sorted and False for unsorted data
	Workload Statistics
SF	Selectivity factor of a query
	Hybrid Layouts Variables
RG_{Size}	Row group size
$Meta_{RG_{Size}}$	Size of meta data for an RG
$Marker_{Size}$	Size of sync marker

Table 2: Parameters of the cost model

Our cost model for hybrid layouts relies on a wide range of statistical information that are summarized in Table 2, containing system constants, data statistics, workload statistics as well as hybrid layout variables. We assume that the constants which depend on the configuration of the environment (e.g., BW_{Disk}, BW_{Net}) are provided. Furthermore, we discuss the collection of statistics (e.g., dataset and workload) in Section 4.

$$Used_{RowGroups} = \frac{(ColValue_{Size} \cdot |T| + Marker_{Size}) \cdot \#Cols}{RG_{Size}} \tag{1}$$

$$|RG| = \frac{|T|}{Used_{RowGroups}} \tag{2}$$

$$TotalMeta_{Size} = (Meta_{RG_{Size}} \cdot \#Cols) \cdot Used_{RowGroups} \tag{3}$$

3.1 Estimating the selection cost

The selection cost model estimates the number of RGs read from the disk and as well as the total read size. For this, first we need to estimate the total number of RGs using Equation 1, and the number of rows in an RG ($|RG|$) using Equation 2. Further, we also need to estimate the total size of metadata (cf. in Equation 3), which is always read from disk to check the matching RGs. Our selection cost model focuses on two cases as discussed earlier. The first one considers filtering using min-max statistics of each RG, and second one filtering using the dictionary.

$$Read_{RowGroups} = \begin{cases} SF \cdot Used_{RowGroups} + 1 & \text{sorted data} \\ Used_{RowGroups} & \text{unsorted and min-max} \\ (1 - (1 - SF)^{|RG|}) \cdot Used_{RowGroups} & \text{unsorted and dictionary} \end{cases} \tag{4}$$

Filtering using min-max statistics. There are two extreme cases when hybrid layouts use min-max statistics to filter RGs, depending on whether data is sorted or not. If data is completely sorted then the selected data will always be contiguous and we can calculate the total number of read RGs based on the selectivity factor as shown in Equation 4. We add one to handle the effect of position variation inside the RGs for sorted data, because hybrid layouts read the whole RG even if there is only one matching row. The reason to add one is illustrated in Figure 2. It shows two RGs and each has 5 rows. Let us assume that we select 3 rows. There are two possible scenarios: (A) there is no overlap and only one RG is read from disk; and (B) there is an overlap and two RGs are read. If we take the average of all possible positions of the first selected row in the first RG, it gives approximately $(SF \cdot Used_{RowGroups}) + 1$.

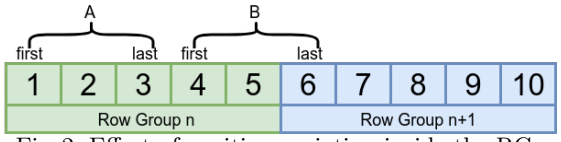

Fig. 2: Effect of position variation inside the RGs

If data is completely unsorted (i.e., uniform distribution), it is unlikely (shown in Section 5.3) to skip any RG, because the distribution of data makes the min-max range of each RG too wide. Hence, the read RGs will be the same as the total number of RGs. We will also experimentally show in Section 5.3 the ineffectiveness of min-max statistics for uniformly distributed unsorted data. Intermediate cases exist for different kinds of skewness, and Equation 4 could be enriched with corresponding estimations without affecting the rest of the paper.

Filtering using the dictionary. The dictionary can also be used to filter RGs when data is encoded. When min-max statistics fail to filter any RG, the dictionary is still very useful, because it contains all existing values. The number of RGs required to be read from disk can be estimated as in Equation 4 (borrowed from bitmap indexes [5]).

$$Used_{Chunks} = \left\lceil \frac{Used_{RowGroups} \cdot RG_{Size}}{Chunk_{Size}} \right\rceil \tag{5}$$

$$Read_{Size} = (Read_{RowGroups} \cdot RG_{Size}) + (TotalMeta_{Size} \cdot Used_{chunks}) \tag{6}$$

$$|Chunk| = \left\lfloor \frac{Chunk_{Size}}{RG_{Size}} \right\rfloor \tag{7}$$

$$Chunk_{Seeks} = \begin{cases} \frac{Read_{RowGroups}}{|Chunk|} + 1 & \text{if sorted} \\ Used_{Chunks} \cdot \left(1 - (1 - \frac{Read_{RowGroups}}{Used_{RowGroups}})^{|Chunk|}\right) & \text{if unsorted} \end{cases} \tag{8}$$

$$W_{ReadTransfer} = \frac{Time_{Disk} + (1-p) \cdot Time_{Net}}{Time_{Seek} + Time_{Disk} + (1-p) \cdot Time_{Net}} \tag{9}$$

$$Query_{Cost} = \frac{Read_{Size}}{Chunk_{Size}} \cdot W_{ReadTransfer} + (Chunk_{Seeks} + Used_{Chunks}) \cdot (1 - W_{ReadTransfer}) \tag{10}$$

The above equations give the expected number of RGs being read from disk, which helps in estimating the total query cost. In distributed processing engines, the data is processed in multiple tasks in parallel and the number of tasks equals to the number of chunks used to store the data, which can be estimated using Equation 5.

Moreover, we observed that each task reads all the metadata separately. The reason is that the distributed processing engines (such as Hadoop and Spark) create a separate process for each task with its own memory. This memory is not accessible to other tasks and hence, forces to read all metadata, and consequently, increases the reading size. We consider this in Equation 6, where we estimate the total read size.

Additionally, we take into consideration the disk seek cost, which depends on the number of chunks being read and also on the number of seeks required to fetch the metadata. The former is equal to the number of read chunks if data is sorted, because it reads consecutive RGs. In Equation 7, we calculate the total number of RGs inside a chunk, which is used in Equation 8 to estimate the total number of seeks for sorted data. Similar to filtering, we add add one to Equation 7 to handle the effect of position variation of RGs inside chunks. On the other hand, when data is unsorted, number of seeks is directly influenced by the distribution of the read RGs, which are non-consecutive due to fact that any RG can match the predicate independently of its position. Thus, it can be approximated by estimating how many RGs are read from a chunk, which depends on the total number of RGs inside a chunk, again calculated using Equation 7. Similarly, we need to estimate the total seeks for reading metadata. As discussed earlier, typically, metadata is stored in the header or footer sections and one seek is required to locate it on the disk. Additionally, it is always read separately in every task, hence the total seeks of metadata will be equal to the total number of tasks (which is equal to the number of chunks).

In distributed processing engines, sometimes, they require to read the data remotely (for instance, it depends on occupancy of machines and unbalanced distribution of workload) and for it, we use a probability p to indicate the likelihood of chunks being accessed locally (i.e., data shipping through the network is needed to reach the operation executor). This is used in Equation 9 to estimate the weight (to calculate the resources usage) of transferring the chunk data compared to the corresponding seek time. Further, it is used along with the total number of seeks in Equation 10 to estimate the total query cost.

$$|D| = \begin{cases} |C| & \text{for global dictionary} \\ \lceil |C| \cdot (1 - ((|C|-1)/|C|)^{|RG|} \rceil & \text{for local dictionary} \end{cases} \tag{11}$$

$$Dictionary_{Size} = |D| \cdot ColValue_{Size} \tag{12}$$

$$Used_{bits} = \lceil log_2 |D| \rceil \tag{13}$$

$$EncodedCol_{Size} = \begin{cases} Used_{bits} \cdot |T| & \text{for global dictionary} \\ Used_{bits} \cdot |RG| & \text{for local dictionary} \end{cases} \tag{14}$$

3.2 Estimating the size of the dictionary

As discussed earlier, hybrid layouts support dictionary encoding, which helps to encode repetitive values to reduce the size and also to facilitate filtering RGs. There are different implementations of dictionary encoding in different types of hybrid layouts. For instance, CarbonData uses a global dictionary to encode the data, whereas Parquet uses a local dictionary inside every RG. However, these two implementations can be easily handled by the same cost model.

Global dictionary. The size of the dictionary depends on the number of values to store inside, which is the total distinct values (i.e., $|C|$) of a column estimated in Equation 11. The size of the dictionary for one column can be then estimated using Equation 12. Further, the average number of bits required to encode one value are estimated in Equation 13, and used in Equation 14 to estimate the encoded size of the data.

Local dictionary. Similarly, the size of the local dictionary depends on the number of values to be put inside the dictionary of an RG, which is the same as the distinct values of a column inside an RG. We estimate the total number of expected distinct values[11] inside an RG as shown in Equation 11. Next, similar to global dictionary, the average number of bits required to encode one value are estimated in Equation 13, and used further in Equation 14 to estimate the encoded size of the data.

4 ATUN-HL

In this section, we first discuss about the collection of data and workload characteristics. Next, we explain our methodology, which utilizes the cost model to find the optimal sizes for RG and dictionary.

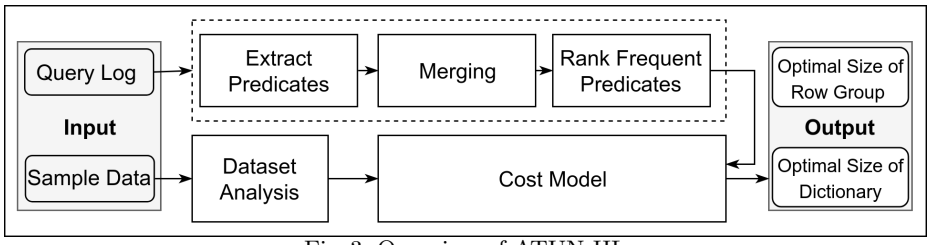

Fig. 3: Overview of ATUN-HL

4.1 Collecting workload and data characteristics

Figure 3 shows the overview of our approach. It takes a query log and the sample data as input, and analyzes them in different components to extract statistical information. The query log is used to extract the information related to the workload. First, our approach extracts the clauses from all the query representatives. Second, it merges the similar clauses or the clauses that can be subsumed. Thirdly, it applies frequent itemset mining approach [8], to rank the most frequent clauses. Finally, it takes the top-k clauses to extract the workload information to be considered. On the other hand, dataset analysis module takes

[11] https://math.stackexchange.com/questions/72223/finding-expected-number-of-distinct-values-selected-from-a-set-of-integers

Algorithm 1: Finding the best size of RG and dictionary

```
1  Possible_DictSizes = {0};
2  for c ∈ Cols do
3  |   DictSize = RoundUpToKiloBytes(EstimateDictionarySize(c));
4  |   Possible_DictSizes.insert(DictSize);
5  end
6  Best = [∞, 0, 0] ;                          // Best[Cost, RG_Size, Dict_Size]
7  for DictSize ∈ Possible_DictSizes do
8  |   Z = EstimateEncodedSize(DictSize);
9  |   Curr_RGSize = Solver(d/dRG_Size (Cost_P(RG_Size, Z)) = 0);
10 |   Curr_Cost = Cost_P(Curr_RGSize, Z);
11 |   if Curr_Cost < Best.Cost then
12 |   |   Best = [Curr_Cost, Curr_RGSize, Dict_Size];
13 end
14 return Best;
```

a sample of data and computes the statistical information listed in Table 2. We use the single column profiling technique from [1].

The use of query log to optimize the parameters for future workloads is justified in [17, 18], which conclude that filters are recurring and only a small portion are entirely new over time.

4.2 Finding the best configuration parameters

Let us assume T is a wide table and has a set of columns defined as $C = \{c_1, c_2, ..., c_n\}$. Similarly, a workload is defined as $Q = \{q_1, q_2, ..., q_n\}$, the frequent clauses extracted from Q are defined as $P = \{p_1, p_2, ..., p_n\}$ the total cost of workload is calculated as $Cost_P(RG_{Size}, Z) = \sum_{p \in P} Query_{Cost}(RG_{Size}, Z)$, where Z represents the total size of T (considering dictionary encoding if needed). Our goal is to minimize $Cost_P$ by selecting the best RG and dictionary sizes.

Algorithm 1 shows the steps to find the optimal sizes of RG and dictionary. It initializes a set in line 1 with the element 0, which corresponds to the scenario where dictionary encoding is completely disabled for all columns. Next, in lines 2 to 4, it iterates over all the columns, computes their dictionary sizes, rounds them up to the nearest kilobytes, and stores them inside the set. Further, in lines 7 to 12, it iterates over all those dictionary sizes and computes the table size according to the current processed dictionary size. Then, the encoded size is used to find the optimal RG size by solving the derivative of the overall cost function. Finally, this value is used to compute the corresponding cost. If the cost is smaller than the best until now, we keep the current processed dictionary and RG sizes as the best ones.

In order to be able to find the minimum cost, we derive the function with respect to the RG size (i.e., $\frac{d}{dRG_{Size}}(Cost_P(RG_{Size}, Z)) = 0$). Equation 15 shows the overall query cost after replacing all variables except read RGs, which still depends on how data has been stored (see Equation 4). Notice that, we need to remove the ceiling function of Equation 5, as well as floor from Equation 7. We

can do the former, because the number of chunks is much smaller than the total number of RGs, and it is only used in calculating the meta size and seek cost, and both are very small compared to the total reading size. Similarly, we can also remove floor in Equation 7, due to its negligible impact on overall cost. We validated their removal with detailed experiments (see Section 5.3).

$$Z = \begin{cases} (ColValue_{Size} \cdot |T| + Marker_{Size}) \cdot \#Cols & \text{no encoding} \\ (Dictionary_{Size} + EncodedCol_{Size} + Marker_{Size}) \cdot \#Cols & \text{encoding} \end{cases}$$

$$Y = Meta_{RG_{Size}} \cdot \#Cols$$

$$\begin{aligned} Query_{Cost}(RG_{Size}, Z) = & \frac{Read_{RowGroups} \cdot RG_{Size} + \frac{Y \cdot Z^2}{RG_{Size} \cdot Chunk_{Size}}}{Chunk_{Size}} \\ & \cdot \quad W_{ReadTransfer} \\ & + \frac{Read_{RowGroups} + \frac{Z}{RG_{Size}}}{\frac{Chunk_{Size}}{RG_{Size}}} \cdot (1 - W_{ReadTransfer}) \end{aligned} \tag{15}$$

5 Experimental Results

In this section, we discuss the setup and the dataset used for our experiments. We also show the ineffectiveness of min-max statistics and usefulness of dictionary for unsorted data. Moreover, we provide the results to validate the accuracy of the cost model and to show the benefits of our approach.

Variable	Value
p	0.97
$Chunk_{Size}$	512MB
BW_{Disk}	1.3×10^8 bytes/second
BW_{Net}	1.25×10^8 bytes/second
$Time_{Seek}$	5.0×10^{-3} seconds
$Meta_{RG_{Size}}$	156 bytes
$Marker_{Size}$	16 bytes

Table 3: Values according to our environment

5.1 Setup

The machine used in our evaluation has a Xeon E5-2630L v2 @2.40GHz CPU, 128GB of main memory, and 1TB SATA-3 of hard disk, and runs Hadoop 2.6.2 and Spark 2.1.10 on Ubuntu 14.04 (64 bit). Our approach is evaluated under two settings: a single node and a 4-machines cluster[12]. In the cluster, we dedicated one machine to HDFS name node and Spark master node together, and the remaining three machines to data nodes for Hadoop and workers for Spark.

We prototyped our approach for Apache Parquet 1.8.2, which further divides each column into multiple data pages (i.e., 1MB) and also stores min-max statistics per data page (i.e., 53 bytes). Nevertheless, currently Parquet does not support data page filtering, so we applied the cost model as described above. If needed, our cost model could be easily adaptabed to data page filtering by

[12]http://www.ac.upc.edu/serveis-tic/altas-prestaciones

simply replacing RG size with data page size and $|RG|$ with the number of rows of a data page.

Table 3 shows the values of all environmental variables in our testbed. In addition, default RG and dictionary sizes in Parquet are 128MB and 1MB, which we use in our evaluation together with best and worse obtained costs.

5.2 Dataset

As mentioned in [4, 12], very wide tables are common in modern analytical systems, because of their advantages in processing compared to normalizing data into narrower tables. Nevertheless, in TPC-H, the widest table has only 16 columns and in TPC-DS[13], only 26. To the best of our knowledge, there is no public benchmark available that consists of wide tables. Hence, we follow [17] to generate a wide table by completely denormalizing all other tables in TPC-H against *lineitem*. The FROM clauses in all queries are consequently changed to the corresponding denormalized table.

5.3 Results

We perform four types of evaluations for our approach. Firstly, we show the drawbacks of min-max based filtering for unsorted data through statistical and also experimental evaluation. Secondly, we show the benefits of dictionary based filtering for unsorted data. Thirdly, we validate the accuracy of our cost model. Finally, we show the performance improvements of our approach on the cluster by comparing it to the baseline setting.

Usefulness of min-max statistics. As previously discussed, min-max statistics are not useful for unsorted data, because uniform data distribution makes it impossible to skip RGs. This behavior is validated with a detailed statistical and experimental evaluation.

$$P_{Skipping} = \frac{\sum_{i=1}^{|C|}\left(\left(\frac{i-1}{|C|}\right)^{|RG|} + \left(\frac{|C|-i}{|C|}\right)^{|RG|}\right)}{|C|} \tag{16}$$

$$Read_{RowGroups} = (1 - P_{Skipping}) \times Used_{RowGroups} \tag{17}$$

Since point queries (i.e., those that search one single value) have higher probability of skipping an RG than the other supported types (namely interval and list of values), and also because of space limitation, we only provide a statistical cost model for this in Equation 16. This estimates the probability of being outside of an RG, which would be the case if the value is less than the minimum of the RG or greater than the maximum. Thus, our cost model adds the probability of both (i.e., minimum and maximum) for each value of that column. Further, the probability of skipping one RG is used in Equation 17, to find the total number of RGs read.

Figure 4a plots Equation 16 for different number of rows $|RG|$, and different number of distinct values of a column $|C|$, which was confirmed with the corresponding experiments. We took 100 as the minimum for $|RG|$, because Parquet

[13] http://www.tpc.org/tpcds

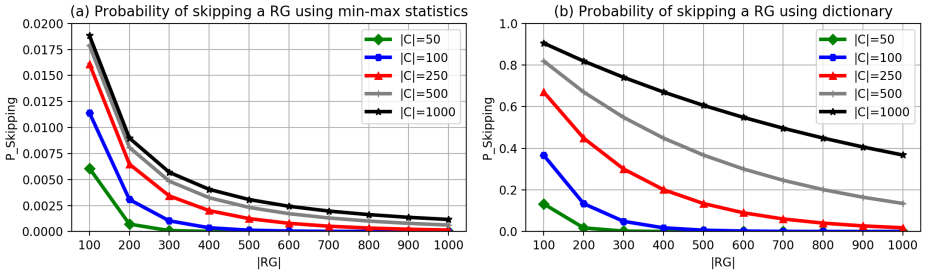

Fig. 4: Probability of skipping one RG

does not allow less rows per RG than that. Thus, it can be observed that the probability of skipping an RG is very low (i.e., always less than $< 2\%$), confirming that min-max statistics are useless for unsorted data. Moreover, when the number of rows in an RG increases, the probability of skipping decreases, which means that it is almost certain that a full scan will be performed. A higher number of distinct values slightly increase the chances of skipping an RG, but it is still very unlikely for RGs with many rows.

Benefits of dictionary encoding. We also plot Equation 4 for dictionary encoding (see Figure 4b), confirming its superiority over min-max statistics. It can be seen that this clearly gives higher probability of skipping, but the chances of skipping still decrease quickly as the number of rows in an RG grows. Yet, it helps with low selectivity queries (when min-max statistics still fail).

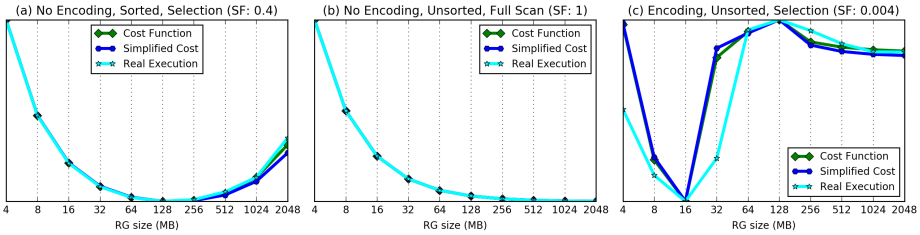

Fig. 5: Comparison between cost model, simplified version, and real execution

Cost model validation. Figure 5 shows the comparison of our cost model, the estimation through its simplified version (which allows derivation as presented in Equation 15), and also actual execution (averaging 250 random runs). We normalized them ($(x-min)/(max-min)$) to facilitate visual comparison. Moreover, as we will show below, the different units (as our cost model only considers I/O cost) do not affect the quality of our prediction to choose the optimal RG size, since the estimated values always preserve the shape of the actual ones (i.e., minimum real cost is obtained for approximately the same value in the model).

We empirically validated the estimations on both sorted and unsorted data, with and without encoding. It can be seen that our cost model and its simplified version are very close and result in approximately the same value. Hence, the derivative can be safely used to find the optimal RG size. Moreover, these both versions follow exactly the same trend as the actual execution.

Performance evaluation. We analyzed TPC-H queries to extract the clauses and ranked them according to their usage. The top 6 clauses which appear in 82% of the queries, are used to find the optimal RG and dictionary sizes. ATUN-HL chooses 30.76MB (that we round up to 32MB) for RG and 1MB for dictionary.

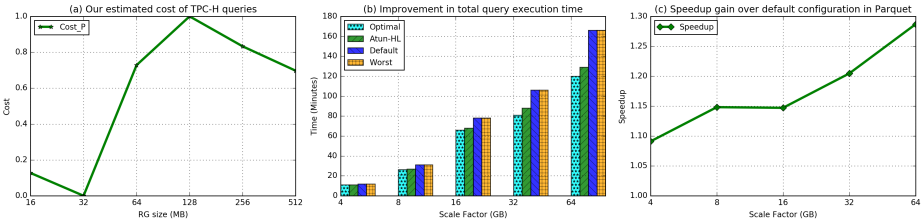

Fig. 6: Speedup gain

Figure 6a shows our estimated overall cost for TPC-H queries. It can be seen that ATUN-HL predicts the default RG size (i.e., 128MB) as the worst configuration (being the minimum at 32MB). As discussed earlier, it is very unlikely for Parquet to skip any RG, when the number of rows in an RG grows. When this turning point is crossed, the larger the RG the better, and our estimated cost depicts this behavior after 128MB. Moreover, we also verified our estimation with detailed experiments as shown in Figure 6b and Figure 6c. Figure 6b compares the time improvements of ATUN-HL against the optimal, default, and worst configurations. ATUN-HL is not far from the optimal configuration, resulting in an 85% of all potential gain. Additionally, Figure 6c shows the relative gain with regard to default RG size, which is 1.2X speedup on average (for the tested scale factors), clearly increasing with the increase in scale factor.

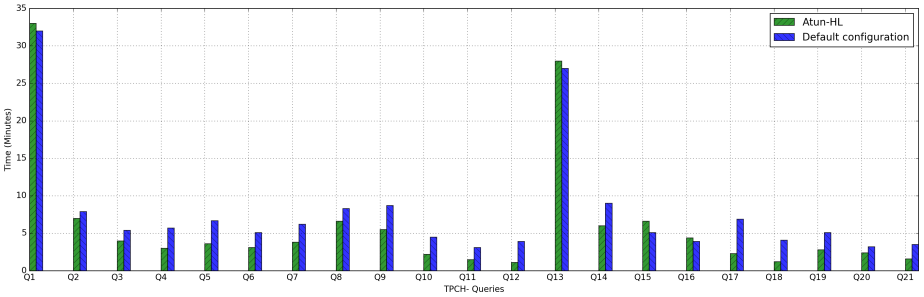

Fig. 7: Improvement in query execution time for 64GB scale factor

Finally, in Figure 7, we also scrutinize the effect on individual query execution time for scale factor $64GB$. This shows that our approach improves the execution time of most of the queries, but does not help those actually performing a full scan (i.e., Q1, Q13, Q15, and Q16) because of one reason (i.e., high SF, $>$ 10%) or another (i.e., string matching using regular expression, which is not yet supported by Parquet). As shown above, the large RG size is always better for full scan.

6 Conclusions

Hybrid layouts are widely used to store processed data in highly distributed Big Data systems to perform ad-hoc analysis. Nevertheless, they have many con-

figurable parameters that need to be tuned according to the characteristics of the data and workload, which can heavily impact query performance. Consequently, we proposed a cost-based approach to help optimizing such hybrid layouts. We prototyped our approach for Apache Parquet, evaluated it on TPC-H queries, and showed the improvement it provides.

Acknowledgement

This research has been funded by the European Commission through the Erasmus Mundus Joint Doctorate "Information Technologies for Business Intelligence - Doctoral College" (IT4BI-DC), and the GENESIS project, funded by the Spanish Ministerio de Ciencia e Innovación under project TIN2016-79269-R.

References

1. Z. Abedjan, L. Golab, and F. Naumann. Data profiling: A tutorial. In *SIGMOD Conference*. ACM, 2017.
2. I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: a hands-free adaptive store. In *SIGMOD Conference*. ACM, 2014.
3. T. Azim, M. Karpathiotakis, and A. Ailamaki. Recache: Reactive caching for fast analytics over heterogeneous data. *PVLDB*, 11(3), 2017.
4. H. Bian, Y. Yan, W. Tao, L. J. Chen, Y. Chen, X. Du, and T. Moscibroda. Wide table layout optimization based on column ordering and duplication. In *SIGMOD Conference*. ACM, 2017.
5. A. F. Cardenas. Analysis and performance of inverted data base structures. *Commun. ACM*, 18(5), 1975.
6. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.
7. M. Ferreira, J. Paiva, M. Bravo, and L. E. T. Rodrigues. Smartfetch: Efficient support for selective queries. In *CloudCom*. IEEE Computer Society, 2015.
8. J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: current status and future directions. *Data Min. Knowl. Discov.*, 15(1), 2007.
9. H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 4(11), 2011.
10. H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, 2011.
11. A. Jindal, J. Quiané-Ruiz, and J. Dittrich. Trojan data layouts: right shoes for a running elephant. In *SoCC*. ACM, 2011.
12. Y. Li and J. M. Patel. Widetable: An accelerator for analytical data processing. *PVLDB*, 7(10), 2014.
13. G. Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *VLDB*, pages 476–487, 1998.
14. R. F. Munir, A. Abelló, O. Romero, M. Thiele, and W. Lehner. A cost-based storage format selector for materialization in big data frameworks. *CoRR*, abs/1806.03901, 2018.
15. R. F. Munir, O. Romero, A. Abelló, B. Bilalli, M. Thiele, and W. Lehner. Resilientstore: A heuristic-based data format selector for intermediate results. In *MEDI*, 2016.
16. K. V. Shvachko. Hdfs scalability: the limits to growth. *Login*, 35(2):6–16, 2010.
17. L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained partitioning for aggressive data skipping. In *SIGMOD Conference*. ACM, 2014.
18. L. Sun, M. J. Franklin, J. Wang, and E. Wu. Skipping-oriented partitioning for columnar layouts. *PVLDB*, 10(4), 2016.