Final publication is available from Mary Ann Liebert, Inc., publishers http://dx.doi.org/10.1089/big.2019.0068

Configuring Parallelism for Hybrid Layouts using Multi-Objective Optimization

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

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

Abstract. Modern organizations typically store their data in a raw format in data lakes. This data is then processed and usually stored under hybrid layouts, because they allow projection and selection operations. Thus allowing (when required) to read less data from the disk. However, this is not very well exploited by distributed processing frameworks (e.g., Hadoop, Spark) when analytical queries are posed. These frameworks divide the data into multiple partitions and then process each partition in a separate task, consequently creating tasks based on the total file size and not the actual size of the data to be read. This typically leads to launching more tasks than needed, which in turn increases the query execution time and induces significant waste of computing resources.
To allow a more efficient use of resources and reduce the query execution time, we propose a method that decides the number of tasks based on the data being read. To this end, we first propose a cost-based model for estimating the size of data read in hybrid layouts. Next, we use the estimated reading size in a multi-objective optimization method to decide the number of tasks and computational resources to be used. We prototyped our solution for Apache Parquet and Spark and found that our estimations are highly correlated (0.96) with the real executions. Furthermore, using TPC-H we show that our recommended configurations are only 5.6% away from the Pareto front and provide 2.1x speedup compared to default solutions.

Keywords: Big data, Hybrid storage layouts, Parallelism, Parquet, Spark

1 Introduction

The size of data is growing exponentially [20, 29]. Since huge volumes of data are difficult to be stored on *model first load later* fashion, organizations end up storing all the the raw data on a distributed file system (e.g., HDFS[3]) or cloud storage (e.g., Amazon S3[4]). In addition, they have their own data pipelines to

[3] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[4] https://aws.amazon.com/s3

process the raw data, and store it into very wide tables [4, 15] using hybrid layouts [3, 16], which have built-in support for projection and selection operations, helping in reading data more efficiently from the disk [27, 28].

There are several available hybrid layout implementations, such as: Optimized Record Columnar (ORC)[5], Parquet[6] and CarbonData[7]. All of them follow the same physical structure as shown in Figure 1. Data is stored into multiple horizontal partitions, known as stripes in ORC, row groups (RGs) in Parquet and blocklets in CarbonData, and each horizontal partition stores its data column-wise, which is beneficial for projection. Statistics about the data are stored in each partition, and they may help on filtering partitions. Furthermore, hybrid layouts support dictionary encoding for compressing repetitive values of individual columns. The dictionary can also be used to filter partitions.

Furthermore, high-level languages (e.g., Apache Pig[8] , Hive[9], etc.) are used to write analytical queries for getting business insights from the processed data. These analytical queries are executed on distributed processing frameworks (such as Hadoop[10] or Spark[11]), which process data in parallel on multiple machines to speed up the analysis. As mentioned above, hybrid layouts allow to read less data from the disk. Yet, this is not thoroughly exploited by distributed frameworks when deciding the number of tasks[12] for processing the data. They always decide the number of tasks based on the total table size and not on the portion of the table being read. This leads to the over-provisioning of tasks, where many tasks remain idle — without any data to process, but still present extra overhead (e.g., initialization time, garbage collection). Furthermore, the idle tasks also waste the computational resources which are assigned to them. The latter is not considered even in the area of cloud computing [12, 13, 21, 24], where computational resources are decided based on the total data size. This leads to wastage of resources and money.

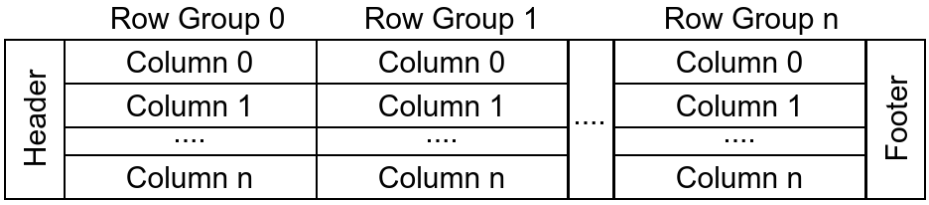

Fig. 1. Structure of hybrid layouts

[5] https://orc.apache.org
[6] https://parquet.apache.org
[7] https://carbondata.apache.org
[8] https://pig.apache.org
[9] https://hive.apache.org
[10] https://hadoop.apache.org
[11] https://spark.apache.org
[12] A task is a unit of work that processes a partition on a machine.

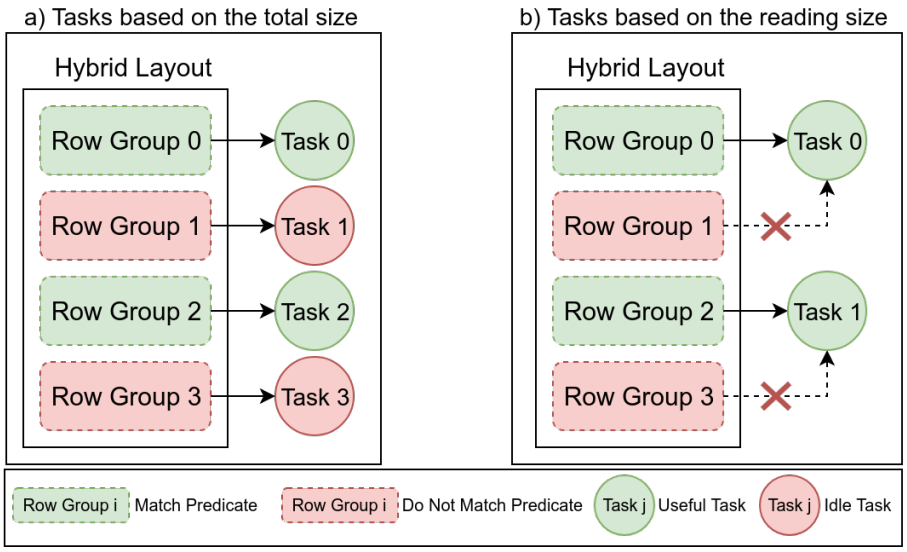

Fig. 2. Parallelism for hybrid layouts in distributed processing frameworks

Motivational Example. Let us assume that we have the processed data stored in hybrid layout, which contains four row groups. Let us further assume that we are executing an analytical query with a filter operation, which only satisfies two RGs. The distributed processing frameworks process the data in parallel by dividing them into multiple partitions (for simplicity, we assume that one partition is equal to a row group). By default a distributed framework would create four tasks. However, two of them would remain idle (c.f. Figure 2a), and yet would read extra metadata from the disk and would require extra initialization time. This would increase the makespan — execution time. Furthermore in terms of computational resources, four executors[13] would be required to execute all these tasks in parallel. Whereas, in an ideal scenario, based on the amount of data read (c.f. Figure 2b) only two tasks with two executors would be enough. The latter would help on saving computational resources and reduce the makespan.

As argued above, we need to decide the number of tasks based on the actual data read from the disk. To do that, we first need to estimate the read size, which can be done by utilizing our cost model presented in [17]. The cost model estimates the scan, projection, and selection sizes for hybrid layouts.

In this paper, we extend it further to estimate the makespan of the query implementing a query based on the estimated reading size. Thus, we design a framework which takes a user query and data statistics as inputs to estimate the reading size, and then through a multi-objective optimization method [10] decide the number of *tasks* and *executors*.

[13] An executor is a computational resource/unit which can execute a task.

After configuring the number of tasks and executors, the query would be automatically submitted to a distributed processing framework. We implemented our approach for Parquet and Spark to show its applicability in real scenarios.

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

- We extend the cost model for hybrid layouts presented in [17] to estimate the makespan of a query.
- We propose a framework based on a multi-objective optimization method [10] that using our extended cost model, configures the number of tasks and executors for a given query.
- We prototype our approach on Parquet and Spark to show its benefits.
- We report the results of our extensive evaluation with TPC-H[14] benchmark.

The remainder of this paper is organized as follows: In Section 2, we discuss the related work. In Sections 3 and 4, we present the cost model and the architecture of our approach. In Section 5, we discuss a multi-objective method to find the number of tasks and executors. In Section 6, we present our experimental results and finally, in Section 7, we conclude the paper.

2 Related Work

Estimating Number of Tasks. There are research works [18, 25] for Hadoop, which estimate the number of mappers and reducers tasks. In [18], the elbow curve technique is used to find the trade-off between number of tasks and execution time. This helps to find the right number of tasks where execution time is minimized. Similarly, [25] utilizes a multi-objective approach for estimating the number of tasks by considering a deadline constraint. These both approaches do not consider the amount of data read, while estimating the number of tasks. These works only estimate the tasks based on the available number of machines and some objectives (such as deadline). As previously argued, the amount of data read is an important factor in deciding the number of tasks.

Resource Provisioning in Cloud. There have been extensive research works [12, 13, 21, 24] by cloud community on resource provisioning. There is also a survey [26] on energy-efficient techniques for big data analytics, which are divided into five categories. One of them (i.e., energy-aware resource allocation) focuses on deciding the number of machines to execute a given query with the aim to save energy. These works from both cloud computing and energy-efficient big data analytics focus more on deciding the number of machines to process an application. They aim at saving energy and computational resources, which indirectly leads to cost savings. However, they make these decisions without considering the reading size. Our approach could help them to decide resource provisioning in more granular level and overall, it can help these works to achieve their goals more efficiently.

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

Tuning Configurable Parameters. There are research works [6, 11, 19] to tune the configurable parameters of distributed processing frameworks. [19] proposes a trial and error approach to tune the configuration parameters of Spark. This work finds the optimal values for these parameters, based on trial and error approach. Similarly, [11] proposes a methodology to profile the impact of different parameters pairs on benchmarking applications, by applying a graph algorithm to create complex candidate configurations. These configurations are checked in parallel and then, the best performing one is chosen. In [8], the shuffle performance in Spark is improved by controlling the total number of shuffle files. This approach consolidated multiple shuffles file into one based on the available cores. This helps in improving the execution time of shuffle phase. [6] profiles the bottlenecks (i.e., JVM, GC, serialization, etc.) of TPC-H queries and parameters are manually configured to avoid the bottlenecks. This significantly increases the query performance.

[2] has proposed a cost model for Spark, which helps to estimate the cost of different query plans and decide the best one. Nevertheless, they assume the number of tasks and executors are fixed. This work is complementary to ours and would optimize the overall query plan, once data is read from disk and available for the first task. As discussed above, these existing works do not explicitly consider the degree of parallelism. Their main aim is to fine tune a cluster of distributed processing framework or find an optimal query plan. Our approach can further help them to improve the query execution time, by configuring the degree of parallelism and computational resources.

3 Cost Model for Hybrid Layouts

In [17], we did not consider configuring the number of tasks and machines, but focused on choosing different storage layouts based on their reading and writing cost. Thus, we extend the cost model to consider new factors (e.g., $Used_{Executors}$, P_{Size}, etc.) and estimations to help in deciding the number of tasks and machines for a given query. In this section, we present the extended cost model for estimating the number of tasks and executors. It should be noted that the number of tasks depends on the partition size (also known as *input split*).

3.1 Parameters of the Cost Model

Our cost model for hybrid layouts relies on a wide range of statistical information that are summarized in Table 1, 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}) are provided. Furthermore, we discuss the collection of statistics (i.e., dataset and workload) in Section 4.

Variable	Description
	System Constants
$Used_{Executors}$	Number of executors for processing
$Chunk_{Size}$	Block size in HDFS
P_{Size}	Size of partition to control the number of tasks
BW_{Disk}	Disk bandwidth
BW_{Net}	Network bandwidth
	Data Statistics
$\|T\|$	Number of rows in a table
$ColValue_{Size}$[1]	Average size of a column value
$\#Cols$	Total columns of T
$Sorted_{Col}$	True for sorted and False for unsorted column
	Workload Statistics
Ref_{Cols}	Number of columns used in an operation
SF	Selectivity factor of an operation
	Hybrid Layout Variables
RG_{Size}	RG size
$Marker_{Size}$	Size of sync marker
$Meta_{Cols_{Size}}$	Size of min-max statistics of columns for an RG
$Body_{Size}$	Size of the body
$Header_{Size}$	Size of the header
$Footer_{Size}$	Size of the footer
$Used_{RGs}$	Number of RGs in the file
$\|RG\|$	Number of rows of a RG

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

Table 1. Parameters of the Cost Model

$$Used_{RGs} = \frac{ColValues_{Size} \cdot |T| \cdot \#Cols}{RG_{Size} - (Marker_{Size} \cdot \#Cols)} \tag{1}$$

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

$$Body_{Size} = ((ColValue_{Size} \cdot |RG| + Marker_{Size}) \cdot \#Cols) \cdot Used_{RGs} \tag{3}$$

$$Meta_{Size} = (MetaCols_{Size} \cdot \#Cols) \cdot \lceil Used_{RGs} \rceil \tag{4}$$

$$Total_{Size} = Header_{Size} + Body_{Size} + Meta_{Size} \tag{5}$$

3.2 Physical Format of Hybrid Layouts

As shown in Figure 1, hybrid layouts divide the data into multiple RGs (estimated using Equation 1) and each RG contains a subset of rows (estimated using Equation 2). In each RG, hybrid layouts store data column-wise and its size can be estimated using Equation 3. Moreover, hybrid layouts also store metadata (e.g., min-max statistics) for each RG inside either the header or footer section, which can be estimated using Equation 4. The size of actual data and metadata are further used in Equation 5 to estimate the total size of the file.

$$Used_{Tasks} = \left\lceil \frac{Body_{Size}}{P_{Size}} \right\rceil \tag{6}$$

$$Used_{Waves} = \left\lceil \frac{Used_{Tasks}}{Used_{Executors}} \right\rceil \tag{7}$$

$$LastWave_{Executors} = ((Used_{Tasks} - 1) \mod Used_{Executors}) + 1 \tag{8}$$

$$\#RGs_{Partition} = \left\lceil \frac{P_{Size}}{RG_{Size}} \right\rceil \tag{9}$$

3.3 Estimating Number of Tasks

Modern distributed processing frameworks decide the number of tasks based on the total file size (which is the size of actual data without metadata) and the partition size (estimated using Equation 6). Moreover, the degree of parallelism depends on the number of executors. All tasks cannot be executed at once, if the number of executors is less than the total number of tasks. Thus, we need multiple rounds/waves to finish the query (estimated using Equation 7). Further, we can calculate the number of executors active in the last wave using Equation 8. Additionally, each partition contains one or more RGs, which can be estimated using Equation 9.

3.4 Estimating MakeSpan

In this paper, we focus on read-only analytical queries, to estimate the amount of data read for their first operation and based on that, we try to find the best partition size to control the number of tasks. Given the simplicity of a file system (far from that of a DBMS), only three operations need to be considered: scan, projection, and selection. These three operations can be generalized to *selection sorted* and *selection unsorted*, because scan and projection operations are just the extreme cases of selection unsorted with selectivity factor of 1 (i.e., they read all RGs).

$$RefCols_{Size} = (ColValue_{Size} \cdot |RG| + Marker_{Size}) \cdot Ref_{Cols} \tag{10}$$

$$Read_{RGs} = \begin{cases} SF \cdot Used_{RGs} + 1 & \text{selection sorted} \\ \left(1 - (1 - SF)^{|RG|}\right) \cdot Used_{RGs} & \text{selection unsorted} \end{cases} \tag{11}$$

Data read estimation. As mentioned above, hybrid layouts help to read only the referred columns and their size can be estimated using Equation 10. Additionally, they use the available metadata (e.g., min-max statistics) to filter some RGs. If selection is applied on sorted data, the average number of read RGs can be calculated directly based on the selectivity factor as shown in Equation 11 (we add one to handle the effect of position variation inside the RGs, because hybrid layouts read the whole RG even if there is only one matching row [16]).

Whereas, for selection of unsorted data, the expected number of read RGs can be estimated using Equation 11 (borrowed from bitmap indexes [5]).

$$Full_{Partitions} = \begin{cases} Used_{Tasks} - 1 & \text{selection unsorted} \\ \left\lceil \frac{Read_{RGs}}{\#RGs_{Partition}} \right\rceil & \text{selection sorted} \end{cases} \quad (12)$$

$$Partial_{Partitions} = \begin{cases} 0 & \text{selection unsorted} \\ 2 & \text{selection sorted} \end{cases} \quad (13)$$

$$Last_{Partition} = \begin{cases} 1 & \text{selection unsorted} \\ 0 & \text{selection sorted} \end{cases} \quad (14)$$

$$Empty_{Partitions} = Used_{Tasks} - Full_{Partitions} - Partial_{Partitions} - Last_{Partition} \quad (15)$$

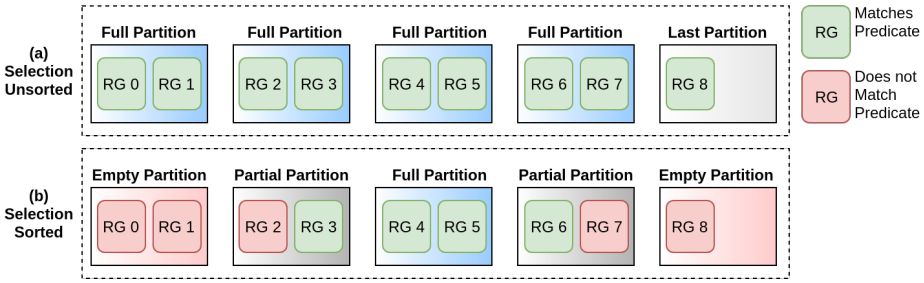

Fig. 3. Type of partitions in selection sorted and unsorted

Types of partitions. Distributed processing frameworks process data by dividing them into multiple partitions, where each partition is processed in a separate task. For *selection unsorted*, every task processes a full partition except the last task, whose partition might not be completely full, as shown in Figure 3a. Equation 12 and Equation 14 indicate the number of full and last partitions. Thus, for unsorted data, any partition has the same probability of containing data. However, *selection sorted* guarantees that we read full partitions, except for, potentially, the first (from where selection starts) and last one (where selection ends), because requested data will not start just at the beginning and finish just at the end of a partition. To reflect this, we always have two partial partitions (Equation 13) and the number of full partitions depends on the number of RGs to be read (Equation 12). Importantly, note that all other partitions will nevertheless read their metadata to determine no data matches the predicate (Equation 15). Figure 3b exemplifies these partitions. It should be noted that the number of partitions and the number of RGs inside each partition are important factors for deciding the correct number of tasks and have direct impact on makespan estimation.

Cost estimation. The total cost of a task depends on four factors: *initialization cost*, *I/O cost*, *CPU cost*, and *networking cost*. The *initialization cost* is constant and can be determined according to the execution environment. The *I/O cost* depends on the amount of data read within a task and the disk bandwidth. We do not consider *CPU cost* due to its negligible impact compared to *I/O cost* (existing works [16, 3] already proved that this is enough to capture the execution trend). Finally, we do not need any shuffling [3], because we focus only on the first operation loading data and therefore, the *networking cost* for shuffling is considered to be zero.

However, there might be some cases when partition size goes beyond the chunk size and it may require some chunks to be transferred over the network. There are two solutions to handle this scenario. The first one is to define a maximum limit on the partition size and always keep it less than the chunk size. The second is to use an existing approach [14], which transfers data in advance to avoid idle cycles on the processing machines. The approach to be used should be chosen based on the business requirements. Our approach would work fine for cloud storage (e.g., Amazon S3), as soon as it accesses the whole file together as an object (not in partitions). Thus, distributed processing frameworks can create a partition without worrying about going beyond the chunk size and data locality.

$$Cost_{Metadata} = \frac{Meta_{Size}}{BW_{Disk}} + \frac{Meta_{Size}}{BW_{Net}} \cdot (Used_{Executors} - 1) \tag{16}$$

There is still a networking cost for metadata (Equation 16), because current solutions require to sequentially transfer metadata to all other executors before start processing the data. Typically, it is read and transferred by the master or driver executor.

$$\begin{aligned}
Cost_{Full_{Partition}} &= Cost_{Init} \\
&\quad + \frac{Meta_{Size} + RefCols_{Size} \cdot \#RGs_{Partition} \cdot (1 - (1 - SF)^{|RG|})}{BW_{Disk}} && (17)\\
Odd_{Data} &= \frac{RefCols_{Size} \cdot (Full_{Partitions} \cdot \#RGsPartition - Read_{RGs})}{Partial_{Partitions}} && (18)\\
Cost_{PartialPartition} &= Cost_{Init} + \frac{Meta_{Size} + Odd_{Data}}{BW_{Disk}} && (19)\\
Residual_{Data} &= RefCols_{Size} \\
&\quad \cdot (Used_{RGs} - \#RGsPartition \cdot Full_{Partitions}) \cdot (1 - (1 - SF)^{|RG|}) && (20)\\
Cost_{Last_{Partition}} &= Cost_{Init} + \frac{Meta_{Size} + Residual_{Data}}{BW_{Disk}} && (21)\\
Cost_{Empty_{Partition}} &= Cost_{Init} + \frac{Meta_{Size}}{BW_{Disk}} && (22)
\end{aligned}$$

Each partition has an initialization cost, which is a constant, and I/O cost (which depends on metadata and the amount of data read inside the partition). As shown in Figure 3, *full partitions* read all matched RGs inside a partition, and

their cost can be estimated using Equation 17. Equation 18 estimates data read from partial partitions and Equation 19 its cost. Equation 20 reads the data left in the last partition and Equation 21 its cost. The other partitions just read metadata and its cost is in Equation 22.

$$\begin{aligned} Cost_{AllTasks} &= Full_{Partitions} \cdot Cost_{Full_{Partition}} \\ &+ Empty_{Partitions} \cdot Cost_{Empty_{Partition}} \\ &+ Partial_{Partitions} \cdot Cost_{Partial_{Partition}} \end{aligned} \tag{23}$$

$$AvgCost_{Task} = \frac{Cost_{AllTasks}}{Used_{Tasks} - Last_{Partition}} \tag{24}$$

These cost of all partitions help to estimate the total cost of all tasks using Equation 23, which is used in Equation 24 to estimate the average cost of a task. It should be noted that the cost of last partition is only applied for selection unsorted and it is considered separately when estimating the total makespan. Thus, we do not consider its cost here.

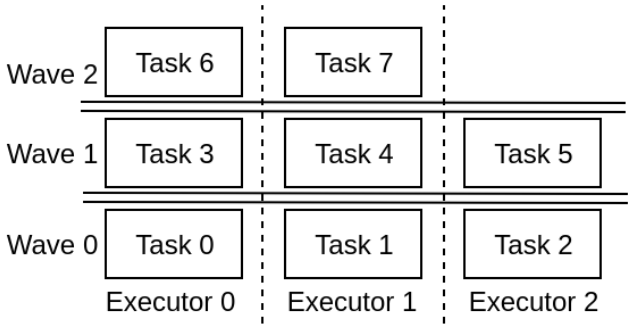

Fig. 4. Execution of tasks

Estimating makespan. As discussed earlier, each task processes different amounts of data and thus, some tasks can finish earlier compared to others. Likewise, each executor can finish their assigned tasks on different times. Thus, we should estimate makespan based on the executor that is processing largest stack of tasks (e.g., in Figure 4, Executor 0 and Executor 1 are the ones with the largest stack). This can be done by estimating standard deviation among tasks and used it further for estimating overall makespan of an operation.

$$Used^{'}_{RGs} = \#RGsPartition \cdot Full_{Partitions} \tag{25}$$

$$Read^{'}_{RGs} = Used^{'}_{RGs} \cdot (1 - (1 - SF)^{|RG|}) \tag{26}$$

For standard deviation, first we need to estimate the number of RGs inside full partitions, using Equation 25. It is further used in Equation 26 to estimate the actual read RGs based on the selectivity factor.

$$Stdev = \begin{cases} \Big(\#RGs_{Partitions} \cdot \frac{Read^{'}_{RGs}}{Used^{'}_{RGs}} \cdot \frac{Used^{'}_{RGs} - Read^{'}_{RGs}}{Used^{'}_{RGs}} & \text{Selection unsorted} \\ \cdot \frac{Used^{'}_{RGs} - \#RGs_{Partitions}}{Used^{'}_{RGs} - 1}\Big) & \\ \sqrt{\frac{\sum_{i=1}^{Used_{Tasks}} (Cost_{Task_i} - AvgCost_{Task})^2}{Used_{Tasks} - 1}} & \text{Selection sorted} \end{cases} \tag{27}$$

Finally, we use hypergeometric distribution [22] for *selection unsorted* to estimate the standard deviation of a full partition in Equation 27, based on the read RGs. Hypergeometric distribution estimates the standard deviation of choosing a subset of items without replacement from the total available items. This is similar to our case where we are also trying to select RGs (i.e., $Read^{'}_{RGs}$) from the total RGs (i.e., $Used^{'}_{RGs}$). Similarly, we also estimate standard deviation in Equation 27 for *selection sorted*.

$$MakeSpan = \begin{cases} \textbf{When LastWave}_{\textbf{Executors}} \textbf{ = 1} & \\ Cost_{Full_{Partitions}} * (Used_{W}aves - 1) + Cost_{Last_{Partition}} + Cost_{Metadata} & \\ \textbf{When LastWave}_{\textbf{Executors}} \textbf{ > 1} & \\ (Used_{Waves} \cdot AvgCost_{Task}) + Cost_{Metadata} & \\ + Stdev \cdot \sqrt{Used_{Waves} \cdot 2 \cdot log_e(LastWave_{Executors})} & \end{cases} \tag{28}$$

Finally, we estimate makespan for an operation using Equation 28. There are two scenarios based on the number of executors active in the last wave. In the first scenario, there is only one executor in the largest stack. In this case, the last task is processing $Last_{Partition}$. Then, we do not need to take any standard deviation, because there is one single largest stack. Thus, we just add the average duration of all task in that stack.

In the second scenario, the makespan depends on metadata transfer, the average cost of a task, the number of executors running in the last wave, and their standard deviation. Thus, we need to estimate expected maximum [7] of those by using the standard deviation as presented in Equation 28, which accounts for the standard deviation of the addition of tasks (i.e., $\sqrt{Used_{Waves}}$), as well as the maximum among executors in the last wave (i.e., $\sqrt{2 \cdot log_e(LastWave_{Executors})}$).

4 Our Approach

In this section, we discuss our approach in detail. Figure 5 shows its architecture, which does not require any change in a distributed processing framework (i.e.,

it is fully transparent for users). The main function blocks of our architecture are the following ones:

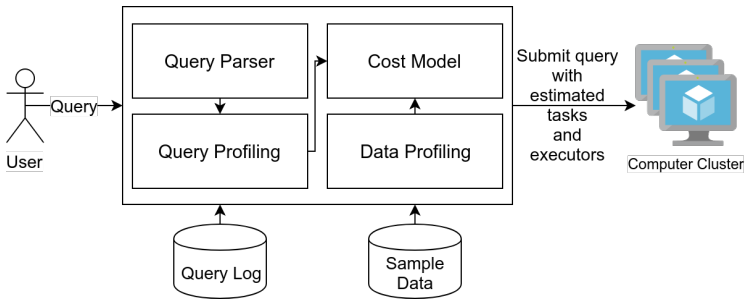

Fig. 5. Architecture of our approach

4.1 Query Parser

The query parser takes a query as input and uses an existing parser (i.e., Spark-SQL parser[15]) to validate its syntax. After validation, it generates the physical plan of the query as an XML and forwards it to the next module. The physical plan represents a tree that starts from input sources to the final output. It also highlights the operations, which can be pushdown to the storage layer.

4.2 Query Profiling

The query profiling takes physical plan as an input and extracts pushdown operations from the plan. Hybrid layouts can only pushdown two operations: projection and selection. It is easy to extract referred columns from the physical plan. Whereas, for selection, it is not possible to extract selectivity factor (SF) from the physical plan. To extract SF, query log needs to be parsed for analyzing the old executions of the same query. Finally, this module passes the pushdown operations along with required statistical information of operations to the cost model.

4.3 Data Profiling

The data profiling module takes a sample of data and computes the statistical information listed in Table 1. We rely on an existing approach, namely single column profiling technique from [1].

[15] https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-AstBuilder.html

4.4 Cost Model

The cost model is used to estimate the reading size for a given query. Typically, a query can have many operations linked together as a Directed Acyclic Graph (DAG). The operations are ordered based on their possibility of pushdown to the storage layer. Hence, the first operation is always a pushdown operation, which reads directly from the disk and impacts parallelism. The subsequent operations takes processed data from the first operation, which modern processing frameworks (e.g., Spark) always keep in memory.

The cost model takes a pushdown operation, workload, data statistics, and cluster configuration as inputs, which are used to estimate the makespan for a given partition size and the number of executors as presented in Section 3. Our goal is to find the best partition size and the number of executors, which can be done using a multi-objective optimization method describe in the following section.

5 Multi-Objective Optimization

In this paper, we focus on optimizing two objectives, which are contradicting to each other. These objectives are *makespan* of query and *resource usage* (i.e., number of executors) required to run the query. We would like to minimize both together. However, they are mutually contradicting, i.e., if we want to reduce makespan, we require more computational resources. In the same way, if we want to save computational resources, we have to compromise makespan. Thus, we need to find a trade-off between them that satisfies user requirements and constraints.

The **first objective function** (i.e., $MakeSpan(Operation_{Type}, P_{Size}, Used_{Executors})$) is based on the makespan estimation according to Equation 28 (as defined in Section 3) for a given operation type, partition size, and the number of executors. Similarly, the **second objective function** (i.e., $Resource_{Usage}(P_{Size}) = Cost_{AllTasks}$ as defined in Equation 23 estimates the resource usage, which increases with the number of tasks.

$$P_{Size} >= RG_{Size} \quad and \quad P_{Size} <= \frac{Total_{Size}}{Used_{Executors}} \tag{29}$$

$$P_{Size} <= Executor_{Memory_{Size}} \tag{30}$$

$$Used_{Executors} <= Max_{Executors} \tag{31}$$

To avoid unfavorable or even impossible configurations, we need to add three constraints. Firstly, Equation 29 guarantees that the partition size is always greater than or equal to the RG size and at the same time, we have enough partitions to utilize all assigned executors as shown in Equation 30. Finally, Equation 31 enforces the maximum number of executors.

Typically, there is no single optimum in a multi-objective optimization problem, but a Pareto front which contains many potentially optimal solutions depending on user prioritization of one objective or another (as shown in Figure 6a).

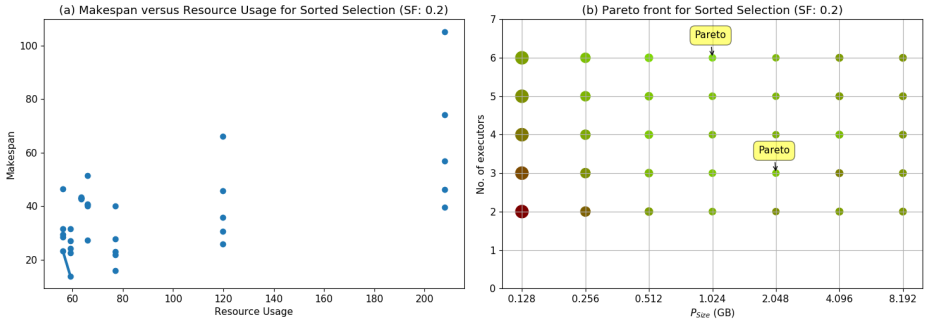

Fig. 6. Pareto front for a selection (circle size represents resource usage, the bigger the more resources; and color represents makespan, red for high and green for low)

Thus, the user has to choose one configuration from the Pareto front to, in the end, execute the query at hand. Our framework[16] facilitates the user choice by reducing the many possible configurations to very few (belonging or close to the Pareto front), so helping her to select one according to her preferences. As shown in Figure 6b, the position in the solution space does not determine the position in the configuration space, which hinders user's choice. In this case, our framework leaves only two (out of thirty-five possible solutions), which satisfy both objectives according to our estimations. When the user selects one of those two, the framework submits the query seamlessly to a processing engine by configuring the partition size and number of executors accordingly.

In this paper, we do not focus on proposing a new multi-objective method, rather we focus on finding the best possible configuration (i.e., number of tasks and executors) for a given query. Thus, we use an existing multi-objective optimization approach, namely NSGA-II [10], implementing genetic algorithms. It simply takes objective functions along with constrains as input, and produces the Pareto front as an output.

6 Experimental Results

In this section, we discuss the setup and dataset used in our experiments. We also provide the results that validate the accuracy of the cost model and show the benefits of our approach.

6.1 Setup

We perform experiments on 5-machines cluster. Each machine has a Xeon E5-2630L v2 @2.40GHz CPU, 128 GB 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). In the cluster, we dedicated one machine for the HDFS name node and Spark

[16] http://www.essi.upc.edu/dtim/tools/adbis2019

Variable	Value
$Used_{Executors}$	2, 3, 4, 5, and 6
$Chunk_{Size}$	128 MB
BW_{Disk}	1.3 x 10^8 bytes/second
BW_{Net}	1.25 x 10^8 bytes/second
$Cost_{Init}$	1 second
RG_{Size}	128 MB
$Marker_{Size}$	16 bytes
$Meta_{Cols_{Size}}$	156 bytes
$Header_{Size}$	4 bytes

Table 2. Values according to our environment

master node together, and the remaining machines to data nodes for Hadoop and executors for Spark. We prototyped our approach for Apache Parquet 1.8.2. Table 2 shows the values of all environmental variables in our testbed. We also configured *replication factor* equals to the number of machines to have replicas on every machine thus avoiding chunk transfer in the case of having partition size greater than the chunk size.

We also instantiated our cost model presented in Section 3 for scan, projection, and selection (both sorted and unsorted). *Scan* operation is just a selection unsorted with selectivity factor 1, referring all the columns of the table. Similarly, *Projection* is also a selection unsorted with selectivity factor 1 and based only on the referred columns. For *Selection*, we just need to give selectivity factor and it would work for both.

6.2 Results

As mentioned in [4, 15], very wide tables are common in modern analytical systems, because of their advantages in processing compared to normalizing data into narrower tables. Nevertheless, to the best of our knowledge, there is no public benchmark available that consists of wide tables. Therefore, in this section, we first validate the accuracy of our cost model for makespan with a synthetic dataset of a very wide table. Further, we present the results to show the benefits of our approach to choose the best configuration for queries over the TPC-H denormalized schema.

Cost model validation. We generated a synthetic dataset of a very wide table with 1186 columns with different data types and 32 GB of size. We executed scan, projection with 10 referred columns, and selection with 0.2 selectivity factor to compare the real executions with our estimations. Figure 7 shows that comparison (notice that, we normalized the results, both real and estimation, like $\frac{x - min}{max - min}$ to facilitate visual comparison).

Figure 7a, Figure 7b, and Figure 7c show the results for a scan operation with different number of executors. Similarly, Figure 7d, Figure 7e, and Figure 7f show

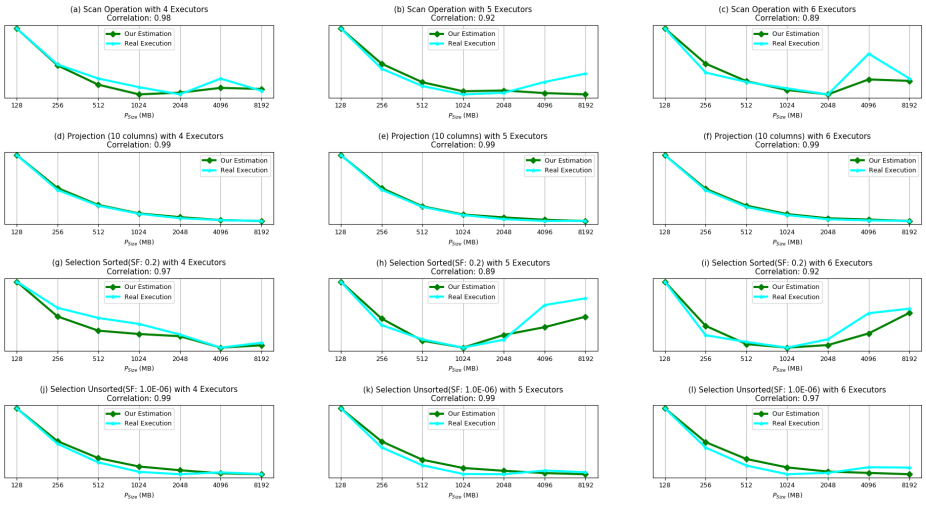

Fig. 7. Validation of our estimation for makespan

the results for a projection operation with different number of executors. Finally, Figure 7g, Figure 7h, and Figure 7i show the accuracy of our estimations in comparison with the real executions for selection operation against sorted data. And, Figure 7j, Figure 7k, and Figure 7l shows the results for selection operation against unsorted data. Observe that, our estimations successfully capture the trends of real executions in almost all cases. Most of our predictions closely follow the real trends. In case of Figure 7c, 7h, and 7i the divergences with the real trend are due to the different units used in our estimation. Yet, the trends are predicted correctly and suffice to find the optimal partition size. The only exception is 7b, where we estimated a lower cost for large partition. Nevertheless, even in this case, our choice is still better than the default partition size.

We also confirm the accuracy of our estimations with the real executions using statistical correlation, which measures how well the cost model estimates are related to the real execution. In Figure 7, it can be seen that our estimations are highly correlated (i.e., overall Pearson correlation coefficient 0.96) to real executions.

Query	SF	Ref Cols	Similar Queries
Q1	[0.98, 0.98]	[7, 7]	-
Q3	[0.0026, 0.0056]	[5, 7]	Q8, Q12, Q17
Q10	[0.011, 0.031]	[4, 11]	Q4, Q5, Q6, Q7, Q11, Q14
Q16	[0.04, 0.08]	[2, 8]	Q2, Q13, Q15, Q18
Q20	[0.000025, 0.0007]	[5, 11]	Q19, Q21
Q22	[0.11, 0.2]	[2, 7]	Q9

Table 3. Representative queries of TPC-H

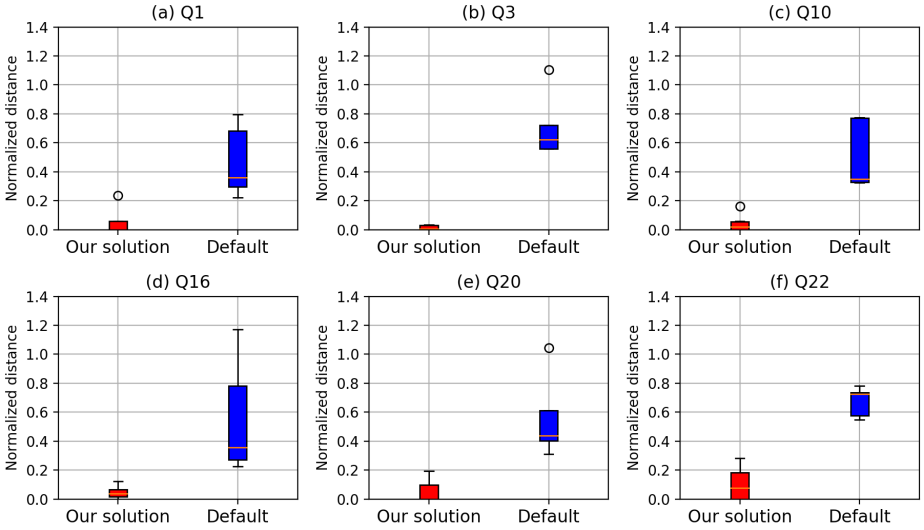

Fig. 8. Comparison between our configurations and default ones for TPC-H

Performance evaluation. In TPC-H, the widest table has only 16 columns and in TPC-DS[17], only 26. Hence, we follow [23] 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. This results, for a scale factor $16GB$, in a denormailized table of $124GB$ being generated for the evaluation. We have chosen six representative queries based on different projected attributes and selectivity factors as shown in Table 3. The table shows the intervals of selectivity factor and number of referred columns of each group of queries. The other queries follow similar access patterns to those selected.

As presented earlier, there is no optimal solution in a multi-objective optimization, but there are many best solutions referred to as Pareto front. The Pareto front of our estimation is denoted as $Pareto_{Estimated}$, and the Pareto front of the real execution is denoted as $Pareto_{Real}$. It could happen that in the $Pareto_{Estimated}$, we miss some real Pareto solutions. These are referred to as $Pareto_{Missed}$. Furthermore, we have the default set of solutions — when a default partition size (i.e., 128MB) is used, denoted as $Default_{Real}$. Finally, each solution has two metrics based on our objectives, namely, *makespan* and *resource usage*.

We compute the Euclidean distance between $Pareto_{Estimated}$ and $Pareto_{Real}$ (both *makespan* and *resource usage* components are normalized to mitigate differences in the scales, resulting on a maximum distance of $\sqrt{2}$), and also to penalize the missed solutions, we compute the distance between $Pareto_{Miss}$ and $Pareto_{Estimated}$ — all these distances compute to *Our Solution*. Furthermore, we

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

also compute the distance between $Default_{Real}$ and $Pareto_{Real}$ — which are represented as *Default Configurations*. More precisely, the Euclidean distance is computed between each solution of one set and all the solutions of the other set, and each time the minimum distance between them is taken.

In Figure 8, we show the Boxplot of the distances corresponding to *Our Solution* compared to the boxplot of the distances to *Default Configurations*. We are showing the results of the representative queries (chosen based on their referred columns and selectivity factors) of TPC-H. Observe the boxplots in *Our Solution* are smaller and closer to zero distance, which means that the solutions proposed (i.e., $Pareto_{Estimated}$) are much closer to the real Pareto front (i.e., $Pareto_{Real}$) than the default configurations (i.e., $Default_{Real}$). In summary, on average we are as close as 5.6% to the real Pareto front, whereas, the default configuration is much further from the Pareto front (on average 58.2%).

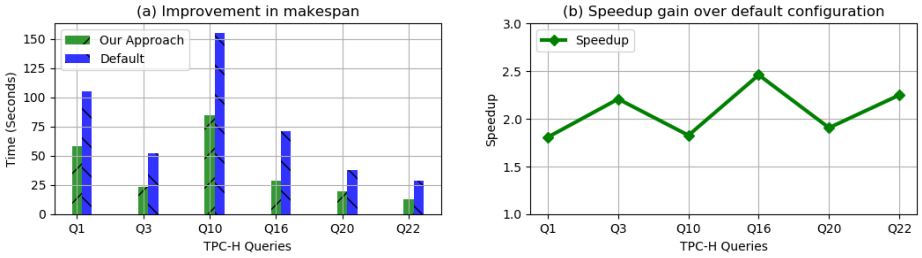

Fig. 9. Speedup gain for TPC-H queries

We also compare the query execution time (i.e., makespan) of our approach with the default configuration (e.g., default partition size of 128MB). As mentioned earlier, we have multiple solutions for a query and we took the minimum makespan among these solutions for comparison. Similarly, we have multiple default configurations and we took the average of their makespans. Figure 9a compares the makespan of TPC-H queries, which highlights the advantage of our approach over the default solutions. Likewise, we also present the speedup gain in Figure 9b, which is between 1.8x to 2.5x. On average, our approach provides 2.1x speedup against the default configuration.

7 Conclusions

Hybrid layouts are widely used to store processed data in highly distributed Big Data systems to perform ad-hoc analysis. These Big Data systems process data on a computers cluster by creating multiple tasks. Typically, they create tasks based on the total size of the table, rather than based on the reading size of the query. Moreover, always using the default configuration has a heavy impact on performance. Thus, we proposed a cost-based framework which utilizes a multi-objective approach to decide the number of tasks and executors for a given query

based on the reading size. 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).

References

1. Z. Abedjan, L. Golab, and F. Naumann. Data profiling: A tutorial. In *SIGMOD Conference*. ACM, 2017.
2. L. Baldacci and M. Golfarelli. A cost model for Spark SQL. *TKDE*, 31(5):819–832, 2019.
3. H. Bian, Y. Tao, G. Jin, Y. Chen, X. Qin, and X. Du. Rainbow: Adaptive layout optimization for wide tables. In *ICDE*, pages 1657–1660, 2018.
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*, 2017.
5. A. F. Cardenas. Analysis and performance of inverted data base structures. *Commun. ACM*, 18(5):253–263, 1975.
6. T. Chiba and T. Onodera. Workload characterization and optimization of TPC-H queries on Apache Spark. In *ISPASS*, pages 112–121, 2016.
7. G. Dasarathy. A simple probability trick for bounding the expected maximum of n random variables. Technical report, Arizona State University, 2011.
8. A. Davidson and A. Or. Optimizing shuffle performance in Spark. Technical report, UC Berkeley, 2013.
9. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.
10. K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evolutionary Computation*, 6(2):182–197, 2002.
11. A. Gounaris and J. Torres. A methodology for Spark parameter tuning. *Big Data Research*, 11:22–32, 2018.
12. H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *SOSP*, page 18, 2011.
13. M. T. Islam, S. Karunasekera, and R. Buyya. dSpark: Deadline-based resource allocation for big data applications in apache spark. In *e-Science*, pages 89–98, 2017.
14. P. Jovanovic, O. Romero, T. Calders, and A. Abelló. H-WorD: Supporting job scheduling in Hadoop with workload-driven data redistribution. In *ADBIS*, pages 306–320, 2016.
15. Y. Li and J. M. Patel. WideTable: An accelerator for analytical data processing. *PVLDB*, 7(10), 2014.
16. R. F. Munir, A. Abelló, O. Romero, M. Thiele, and W. Lehner. ATUN-HL: Auto tuning of hybrid layouts using workload and data characteristics. In *ADBIS*, pages 200–215, 2018.

17. 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.
18. P. P. Nghiem and S. M. Figueira. Towards efficient resource provisioning in MapReduce. *JPDC*, 95:29–41, 2016.
19. P. Petridis, A. Gounaris, and J. Torres. Spark parameter tuning via trial-and-error. In *INNS*, pages 226–237, 2016.
20. K. V. Shvachko. HDFS scalability: the limits to growth. *Login*, 35(2):6–16, 2010.
21. S. Sidhanta, W. M. Golab, and S. Mukhopadhyay. Optex: A deadline-aware cost optimization model for Spark. In *CCGrid*, pages 193–202, 2016.
22. M. Skala. Hypergeometric tail inequalities: ending the insanity. *CoRR*, abs/1311.5939, 2013.
23. L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained partitioning for aggressive data skipping. In *SIGMOD*, pages 1115–1126, 2014.
24. L. Thai, B. Varghese, and A. Barker. Budget constrained execution of multiple bag-of-tasks applications on the cloud. *CoRR*, abs/1507.05467, 2015.
25. A. Verma, L. Cherkasova, and R. H. Campbell. Resource provisioning framework for MapReduce jobs with performance goals. In *USENIX*, pages 165–186, 2011.
26. W. Wu, W. Lin, C. Hsu, and L. He. Energy-efficient Hadoop for big data analytics and computing: A systematic review and research insights. *Future Generation Comp. Syst.*, 86:1351–1367, 2018.
27. Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, Z. Xu RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems *ICDE*, pages 1199–1208, 2011.
28. T Ivanov, M. Pergolesi The impact of columnar file formats on SQL-on-hadoop engine performance: A study on ORC and Parquet *Concurrency Computat Pract Exper*, 32:e5523, 2020.
29. D. Reinsel, J. Gantz, J. Rydning The Digitization of the World - From Edge to Core. *IDC*, 2018.