ResilientStore: A Heuristic-based Data Format Selector for Intermediate Results

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

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

Abstract. Large-scale data analysis is an important activity in many organizations that typically requires the deployment of data-intensive workflows. As data is processed these workflows generate large intermediate results, which are typically pipelined from one operator to the following. However, if materialized, these results become reusable, hence, subsequent workflows need not recompute them. There are already many solutions that materialize intermediate results but all of them assume a fixed data format. A fixed format, however, may not be the optimal one for every situation. For example, it is well-known that different data fragmentation strategies (e.g., horizontal and vertical) behave better or worse according to the access patterns of the subsequent operations. In this paper, we present ResilientStore, which assists on selecting the most appropriate data format for materializing intermediate results. Given a workflow and a set of materialization points, it uses rule-based heuristics to choose the best storage data format based on subsequent access patterns. We have implemented ResilientStore for HDFS and three different data formats: SequenceFile, Parquet and Avro. Experimental results show that our solution gives 18% better performance than any solution based on a single fixed format.

Keywords: Big data, data-intensive workflows, intermediate results, data format, HDFS

1 Introduction

Large-scale data analysis is an important activity for many organizations. It is typically performed by deploying pipelined workflows (known as Data Intensive Workflows (DIW)) on Hadoop[3] clusters. Many high-level languages (namely as Hive[4] and Pig[5]) have been introduced to facilitate the execution of analysis tasks on Hadoop. These languages aim at decomposing the tasks into multiple pipelined MapReduce [4] jobs. Each task produces results that are commonly

[3] https://hadoop.apache.org
[4] http://hive.apache.org
[5] http://pig.apache.org

referred to as intermediate results. Intermediate results are used by multiple subsequent tasks. An in-depth study of MapReduce workloads for seven enterprises [3] shows that 80% of intermediate results are re-accessed in different parts of DIWs. This study demonstrates the importance of materializing the right intermediate results for speeding up the flows that re-access them. However, the study gives rise to two questions as well: "What to materialize?" and "How to materialize?".

Answers to the first question have already been given. In [6, 16], they provide tools which help in choosing what to materialize. However, these solutions do not provide help in terms of how to materialize the intermediate results. They typically store them directly on HDFS [10], where I/O operations are expensive. Hence, unnecessary reads and writes performed, increase the execution costs of DIWs.

Researchers have come up with different data formats that help in reducing the amount of read and write operations. The proposed data formats are built on top of HDFS and are designed for fast loading, fast query processing and efficient storage utilization. Among the most popular formats, we find: Record Columnar File (RCFile) [11], Optimized Row Columnar (ORC)[6], Avro[7], Parquet[8] and SequenceFile[9]. They differ from each other in terms of their layout. Avro, for instance, uses a horizontal layout, whereas Parquet uses a hybrid layout. None of them is the universal best choice; different workloads require different layouts to achieve optimal performance [2].

Note also that, none of the solutions that answer the first question consider different formats and they store intermediate results using a single fixed format. A single fixed format though, is not appropriate for all types of workloads[10] and this is already identified in previous works [2, 13, 9]. These works clearly demonstrate the importance of the second question. Ideally, the materialized results should be stored in the most appropriate format for their later reuse.

We contend that a properly chosen data format reduces the load time of the intermediate results in their subsequent use, and overall, reduces the execution cost of DIW. That is why, in this paper we present ResilientStore, which decides the most appropriate data format for intermediate results using heuristic rules and considering the subsequent access characteristics. The contributions of this paper are as follows:

- We show the performance bottleneck of data formats in different workloads.
- We define a set of heuristic rules to select the appropriate data format based on the access characteristics of the workloads.
- We show that by using ResilientStore in selecting the data format we can reduce the load time of the intermediate results.

[6] https://orc.apache.org
[7] http://avro.apache.org
[8] http://parquet.apache.org
[9] http://wiki.apache.org/hadoop/SequenceFile
[10] http://www.svds.com/how-to-choose-a-data-format

The remainder of this paper is organized as follows. In Section 2, we discuss the Hadoop data formats and provide the motivating example of our work. In Sections 3 and 4, we discuss the heuristic rules, the architecture of ResilientStore and its implementation. In Section 5, we show our experiment results. In Section 6, we have a discussion of the related work. Finally in Section 7, we conclude the paper and discuss our future work.

2 Background and Motivating Example

In this section, we discuss different storage layouts and their Hadoop representative formats. Moreover, we present an example to show the performance of different formats for different workloads. This example motivates our work and shows the importance of considering different formats for the materialization of intermediate results. Note that, in the context of this paper, layout refers to the fragmentation strategy (e.g., horizontal, vertical) and format refers to the file format (e.g., Avro, Parquet, SequenceFile).

2.1 Hadoop Data Formats

One of the most commonly used formats in Hadoop is the raw format. Despite its extensive use, it suffers from many problems. For instance, it is not splittable after compression or it does not store schema information. Different binary formats have been proposed in order to overcome these problems [9] and they can be classified according to the layout they follow: horizontal, vertical or hybrid.

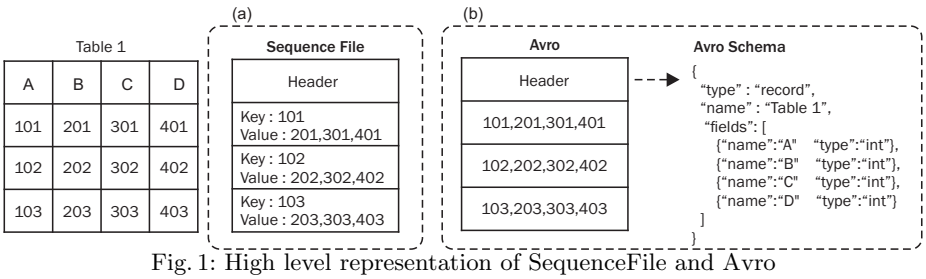

Fig. 1: High level representation of SequenceFile and Avro

Horizontal Layout. A format implementing a horizontal layout organizes data in rows. These formats excel when workloads require full scans. If all the columns are not required, they perform unnecessary reads from disk, since non-required columns will be read anyway. However, these formats are good for data insertion. In Hadoop, the SequenceFile and Avro formats implement a horizontal layout. Figure 1 shows an example table and its corresponding format in SequenceFile (i.e., Figure 1a) and Avro (i.e., Figure 1b).

Vertical Layout. Formats following a vertical layout organize data in columns and each row is split into different groups of columns. Each group consists of multiple columns stored together. This kind of structure helps on reading less data when a query requires only a subset of columns (i.e., improves the effective

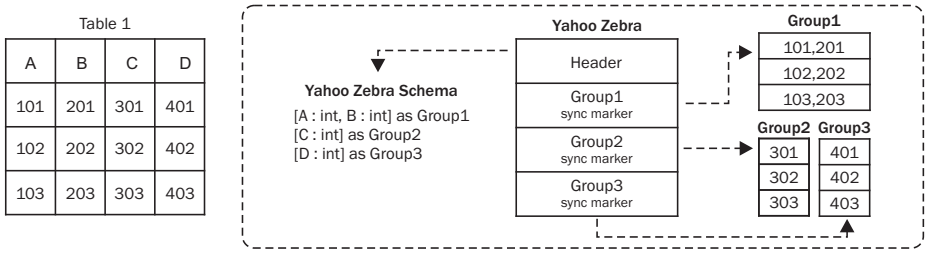

Fig. 2: High level representation of Yahoo Zebra

read ratio). In Hadoop, Yahoo Zebra[11] implements a vertical layout. Figure 2 exemplifies the Yahoo Zebra format.

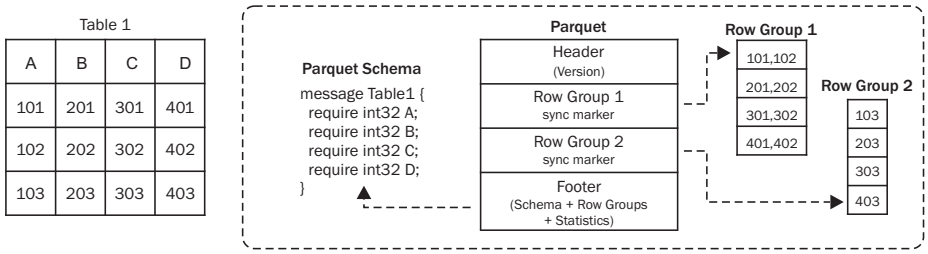

Fig. 3: High level representation of Parquet

Hybrid Layout. A hybrid layout combines the horizontal and vertical layouts. Data stored is divided into row groups and each row inside the row group is further divided into columns. Implementations following a hybrid layout, such as ORC and Parquet, are available for Hadoop. Parquet format is depicted in Figure 3.

Features	**Horizontal**		**Vertical**	**Hybrid**	
	Sequence Files	**Avro**	**Yahoo Zebra**	**ORC**	**Parquet**
Schema	No	Yes	Yes	Yes	Yes
Column Pruning	No	No	Yes	Yes	Yes
Predicate Pushdown	No	No	No	Yes	Yes
Indexing Information	No	No	No	Yes	Yes
Statistics Information	No	No	No	Yes	Yes
Nested Records	No	No	Yes	Yes	Yes

Table 1: Comparison of data formats

In Table 1, a comparison of all the layouts and their representative formats is given. This allows to look at their features side by side. As it can be noted from the table, all formats except SequenceFile, store the schemas of data. The schema information helps during the data serialization and de-serialization phase by avoiding the need to cast the data at the application level - which is a costly

[11] http://pig.apache.org/docs/r0.9.1/zebra_pig.html

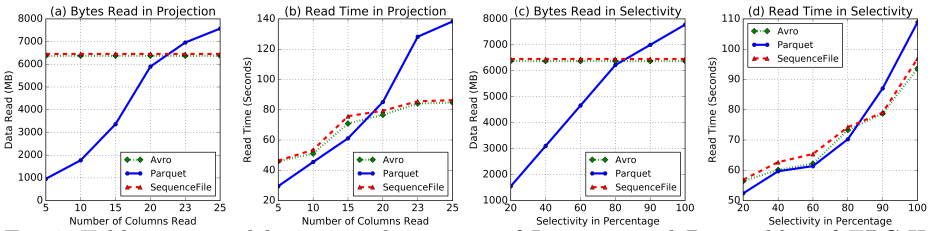

Fig. 4: Table is created by materializing join of Lineitem and Part tables of TPC-H (Scale Factor: 4)

operation. Moreover, the table shows that both vertical and hybrid layouts provide support for column pruning. It means, they only read the required columns and do not perform unnecessary reads. Hybrid layouts can also push down the selection predicates into the storage layer because they store indexing information that helps in filtering the records while reading. Furthermore, since hybrid layouts store statistical information for each column, they enable easier computation of aggregates. Additionally, vertical and hybrid layouts have support for nested records which helps in storing bag, map and custom user data types. It can also be noted that hybrid layouts have more features but they also have a significant overhead when writing and therefore when reading the same amount of data (due to the amount of metadata stored with data).

In summary, these formats have different features that make them perform better or worse for different workloads. Generally, hybrid layouts perform well if a subset of data is read. On the other hand, horizontal layouts perform well if all, or most of the data is read.

2.2 Motivating Example

In this section we present the results drawn by storing the intermediate results of a DIW in different formats. Note that we do not consider the raw format because of the drawbacks mentioned in the previous section. Consequently, nowadays, mostly binary formats are used in real-world scenarios [2]. The results show that the different features of each binary format prevents them from being a universal solution for all types of workloads. Even more now, when mixed workloads are present, consisting of reporting, interactive, or data mining queries. For instance, Parquet is good for reporting and interactive queries whereas Avro is good for data mining queries where most data is read. Different types of queries can be found in different parts of DIWs. As a matter of fact, queries must be analyzed in order to find the proper format for the materialization.

In this example we create a DIW from the TPC-H benchmark[12] queries and materialize in different Hadoop formats a join between the Lineitem and Part tables. In Figure 4, results of our example are shown for SequenceFile, Parquet and Avro. We present now the results of reading such materialization by considering two subsequent operators in the DIW: Projection and Selection. Figures 4a and 4b depict the case when a Projection follows with different numbers of columns to be read from HDFS. In 4a we measure the size of the data read

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

and in 4b we measure the needed time to read the data. Note that Parquet is performing better in the first 20 columns but after that SequenceFile and Avro take the lead. Similarly, Figure 4c and 4d show the same type of experiment but for Selection.

From the results of this example, we can observe that performance depends on the data format and the subsequent operation in the DIW (i.e., the kind of workload). As a matter of fact, this supports our proposed hypothesis that different formats must be chosen depending on the workload characteristics.

3 System Model and Heuristic Rules

This section presents a formal notation of our problem, which we use to define the rules used to decide the data format for materialized intermediate results. As first approach, we opt for heuristics rules. Lightweight approximations have consistently been used in databases before as they yield a good balance between the performance gain obtained and the extra overhead introduced. For example, [6] uses heuristic rules to decide what nodes to materialize given a DIW. The reason is that, unlike cost-based solutions, heuristic rules do not require to gather run-time statistics (e.g., selectivity factor per operation) and thus do not yield any extra performance overhead. Yet, such lightweight approaches can yield significant improvements.

System Model. We formally define a DIW as follows:

$$
\begin{aligned}
&DIW \leftarrow DAG(V, E), \text{ where} \\
&V = \{v_1, v_2, ..., v_n\} \text{ and } E = \{e_1, e_2, ..., e_n\} \\
&M \subseteq V \text{ and } \forall x \in \text{ M, } O(x) \subseteq E \\
&getOP : E \rightarrow \{op_1, op_2, ..., op_n\} \\
&getType : E \rightarrow \{Type_1, Type_2, ..., Type_n\} \\
&getCol_{op} : op \rightarrow \mathcal{P}\{col_1, col_2, ..., col_n\} \\
&getCol_{v} : V \rightarrow \mathcal{P}\{col_1, col_2, ..., col_n\} \\
&getBest : M \rightarrow \{format_1, format_2, ..., format_n\}
\end{aligned}
$$

A DIW is a DAG that consists of vertices (V) and edges (E). A vertex represents a set of data and an edge represents an operator. More precisely, an edge represents an operator applied to the data in its starting vertex. The ending vertex represents the data delivered by the operator after processing the input data. Note that an edge is adorned with schema information; i.e., the columns to which the operator applies. Function **getOP** and **getType** are used to get the instance and type of an operator for a given edge, respectively. Additionally, function $\mathbf{getCol}_{op}$ is used to get the set of columns on which an operator is executed. Similarly, the function $\mathbf{getCol}_{v}$ is used to extract the set of columns of a vertex. In the above notation, M denotes the materialized nodes, which are a subset of V. $O(x)$ represent the outgoing edges from a materialized node, which is a subset of E. Finally, given that our set of rules may decide that more than one format is suitable for a materialized node. We introduce the function **getBest** in

order to choose one among them. This function compares the different formats and chooses the one which has more features.

Heuristic Rules for Format Selection. We now introduce the heuristic rules used to decide what format to choose for a given materialized node. These rules derive from the well-known properties of horizontal, vertical and hybrid layouts and the specific format features presented in Section 2:

$\forall x \in M$

$rule1 : x \rightarrow$ SequenceFile, `IF` $size(getCol_v(x)) = 2$

$rule2 : x \rightarrow$ Parquet, `IF` $\exists e \in O(x)$, `WHERE` $getType(e) \in \{AggregationOps\}$

$rule3 : x \rightarrow$ Parquet, `IF` $\exists e \in O(x)$, `WHERE` $getCol_{op}(getOP(e)) \subseteq getCol_v(x)$

$rule4 : x \rightarrow$ Avro, `IF` $\forall e \in O(x)$, `WHERE` $getCol_{op}(getOP(e)) = getCol_v(x)$

$rule5 : x \rightarrow$ Avro, `IF` $\exists e \in O(x)$, `WHERE` $getType(e) \in \{Join, CartesianProduct, GroupALL, Distinct\}$

`Rule1` chooses SequenceFile for the materialization of nodes that have exactly two columns. This is an immediate application of the SequenceFile format (which stores data as key-value pairs). Otherwise, several columns need to be combined (e.g., with a separator marker such as "-" or ";") either in the key or the value and parsed at the application level. `Rule2` chooses Parquet when performing aggregations on data. Since Parquet stores statistical information for each column, it is the most efficient when computing aggregates. Since Parquet implements a hybrid layout, it is also the best choice when it comes to read subsets of data or when operators apply on subsets of columns (see Table 1). This rationale is behind `rule3`. Oppositely, Avro is chosen when all the data is read or when the operator does not apply on a certain subset of columns. This is a consequence of Avro implementing a horizontal layout. Hence, `rule4` (the operator affects all columns) and `rule5` (the operator requires to read the whole data without filtering) recommend using Avro in these respective cases. The heuristic rules defined are mutually exclusive. They can be applied independently of each other and without any fixed order. Note that, these rules do not consider vertical layouts because they are subsumed by hybrid layouts. However, using our formal notation other formats can be added easily.

After applying the heuristic rules, there exist some cases where multiple choices are suitable. In order to circumvent this problem, we have defined the function `getBest`. This function gives highest priority to Parquet owing to the fact that Parquet has more features and a more flexible behavior. The second highest priority is assigned to Avro because it stores schema information about the data which speeds up the reading. Finally, the lowest rank belongs to SequenceFile which is only chosen for key-value data.

4 ResilientStore

In this section, we first discuss about the materialization of intermediate results. Then, we discuss the architecture of our system and its implementation. Finally, we discuss its format selection algorithm.

4.1 Materialization of Intermediate Results

In previous sections, we mentioned that re-accessing of data occurs very often (i.e., 80% of the time [3]), and there are already available solutions [6, 16] for deciding on the materialization of intermediate results. Hence, we use one of these solutions for the materialization phase, namely, ReStore [6]. However, any other solution can be used as long as it provides the nodes to be materialized. The heuristics used in ReStore are categorized into conservative and aggressive. Conservative heuristics refer to the materialization of the outputs from PROJECT and FILTER operators, because they reduce the size of data. Whereas, aggressive heuristics are used to materialize the outputs of JOIN, GROUP and CoGROUP[13] operators, because they are computation intensive. These heuristics are used to decide about the materialization of the results to be reused by subsequent operators in DIWs. However, note that ReStore does not consider the data format to be used for the materialization. Our approach fills this gap by using the aforementioned heuristic rules for selecting the most appropriate data format when materializing intermediate results.

4.2 System Architecture

In Figure 5, we depict the architecture of ResilientStore. It takes a workflow (DIW) as input and returns a DIW where the nodes to be materialized are tagged and the most appropriate data format is selected. First, ResilientStore applies the ReStore heuristic rules to choose the best nodes for materialization and then applies format selection heuristic rules (Section 3) to decide the most suitable data format.

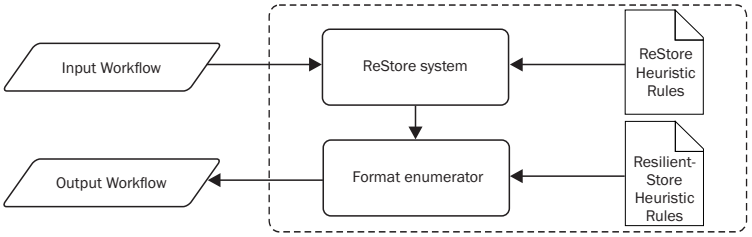

Fig. 5: System Architecture

Our prototype is implemented using Pig as this is the most popular language for executing DIWs on a Hadoop cluster [3] and also because of compatibility reasons with ReStore. Thus, we first instantiated the conceptual operations mentioned in our format selection heuristic rules with Pig Operators. The *Aggregation Ops* (see `rule2`) provided by Pig include SUM, MIN, MAX and COUNT. The set of Pig operations to be considered in `rule5` are JOIN, CROSS (Cartesian Product), COGROUP, GROUPALL and DISTINCT. However, our rules are independent of Pig. We can use any other language (e.g., Apache Drill or Apache Hive) just by instantiating the conceptual operators with the operators of that language.

[13] A Pig operation combining GROUP BY and JOIN

Algorithm 1 ResilientStore data format selection algorithm

```
1: procedure RS–FORMAT(x)
2:     ruleSet = getResilientStoreRules()
3:     formatSet = ∅
4:     for each rule ∈ ruleSet do
5:         format = rule(x)
6:         formatSet.append(format)
7:     end for
8:     bestFormat = getBest(formatSet)
9:     return bestFormat
10: end procedure
```

In order to choose the nodes to be materialized, our solution first applies the heuristic rules of ReStore on each Pig script. Then, once the nodes to be materialized are chosen, the scripts containing them are further analyzed using our heuristic rules to decide the most appropriate data format. Note that, our heuristic rules only consider the first operator of subsequent scripts which are reading these materialized nodes. The reason is that, only the first operator has effect on reading the data from the disk and subsequent operators read the data from the memory. After the data format is decided, the serialization in the selected format needs to be carried out. The serialization process is straightforward for Avro and Parquet because they automatically infer the tuples' schema during the serialization and de-serialization phases. SequenceFile, however, requires explicit key-value pairs for the serialization and de-serialization. Hence, our system automatically converts each tuple into a key-value pair (one attribute is set as key, and the other as value).

4.3 Format Enumerator Algorithm

Our algorithm is shown in Algorithm 1. The algorithm, takes a materialized node as input and finds the best storage format for it. In lines 4 to 7, it iteratively applies all the ResilientStore rules which we have defined in Section 3 and gets their suggested formats. Then, in line 8 it gets the best format among the ones that were suggested. Finally, in line 9 it returns the chosen format. Note that, this algorithm is then iteratively run on every materialized node in order to choose the best format for each one of them.

5 Experiments

This section reports on our experimental findings. Note, first, that we are not considering compression for a fairer comparison between different formats. Second, we are assuming data is uniformly distributed.

5.1 Setup and Dataset

Our experiments are performed on a 8-machine cluster[14]. Each machine has a Xeon E5-2630L v2 @2.40GHz CPU, 128GB of main memory and 1TB SATA-3 of hard disk. Each machine runs Hadoop 2.6.0 and Pig 0.15.0 on Ubuntu 14.04 (64 bit). We have dedicated one machine for the name node and the remaining seven machines for data nodes.

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

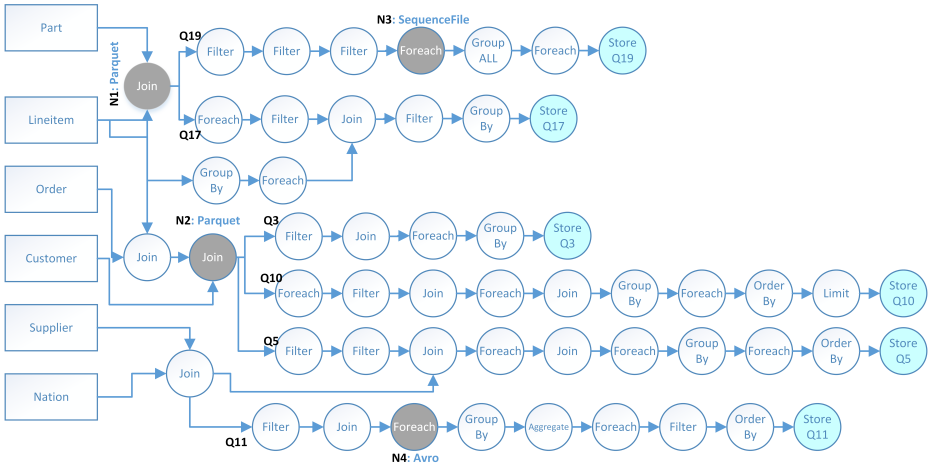

Fig. 6: DIW of six TPC-H queries

We use the TPC-H[12] benchmarking tool to generate datasets and queries. In order to create a complex DIW, we use Quarry [15]. Quarry is used to combine all TPC-H queries into one integrated DIW as shown in Figure 6. We are using TPC-H because it is a standard benchmarking tool and it contains queries which cover all possible cases.

In order to perform more realistic experiments, we generate data with scale factor ranging from 1GB to 128GB. In our experiments, ReStore chooses 8 nodes to be materialized after applying both its aggressive and conservative heuristics. The aggressive heuristics decide the materialization of the output of 6 JOINs and the conservative heuristics decide to materialize that of 2 FOREACH operations. We then apply our heuristic rules (see Section 3) to choose the format of the materialized nodes. For the 6 JOINs, we choose Parquet, while Avro and SequenceFile are chosen, respectively, for the first and second FOREACH. ResilientStore choose the best format in all cases. For discussion, we choose 4 nodes which cover the three formats, as shown in Figure 6. The DIW used in our experiments is available online[15]. Additionally, we choose two metrics to analyze our approach, namely write time and read time, and measure them for each materialized node using the following formulas.

- `write_time = # of HDFS blocks * cost of writing one HDFS block`
- `read_time = (# of HDFS blocks * cost of reading one HDFS block) + execution cost of the first operator of the query`

We only consider the first operator in `read_time` because the subsequent operators are executed in memory and hence they read from memory instead of HDFS.

5.2 Results

This section discusses in detail the four materialized nodes chosen in the previous section. Figure 7 and 8 show the two nodes materialized using Parquet.

[15] http://ranafaisal.info/?attachment_id=153

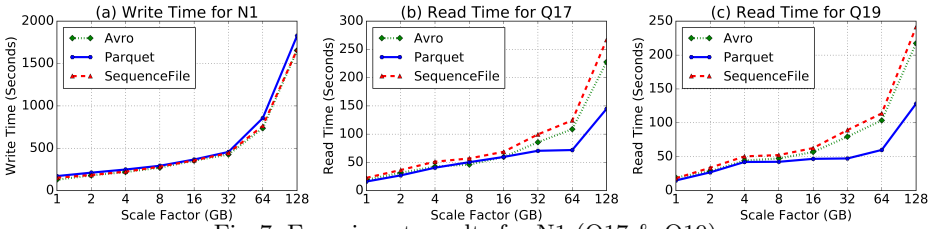

Fig. 7: Experiment results for N1 (Q17 & Q19)

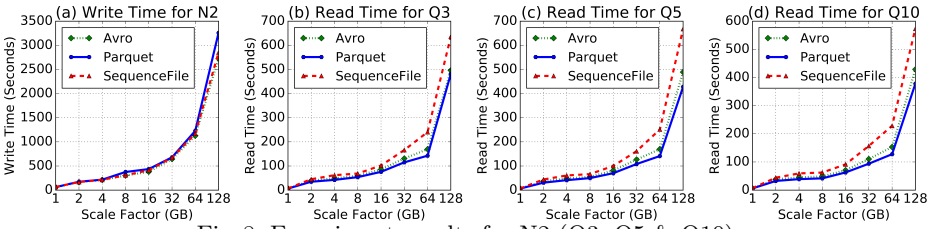

Fig. 8: Experiment results for N2 (Q3, Q5 & Q10)

Note that Parquet spends more time in writing (since it writes more metadata; see Section 2) but performs much better in reading (since it predicates to the storage layer). Figure 7 (b and c) and 8 (b, c and d) show the reading time for the intermediate results in different queries. The metadata writing overhead (e.g., schema) proves beneficial when reading is performed.

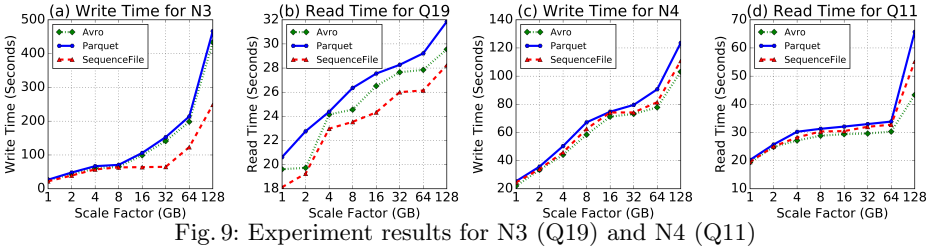

Fig. 9: Experiment results for N3 (Q19) and N4 (Q11)

Furthermore, as it can be noted from Figure 9a and 9b, SequenceFile is a better choice for N3. For all the other nodes SequenceFile takes more time in writing than Avro, because it stores data as key-value pairs and for columns stored in the value it needs markers to separate them. However, in N3 SequenceFile writes less data since two columns are written (one as key, the other as value) and no marker is necessary. In N3, SequenceFile performs also good when reading because it does not need to convert key-value pairs back to tuples.

In Figure 9c and 9d, we show the performance for N4, which chooses Avro. Note that Avro writes less data for all nodes except for N3. This is the reason why Avro performs well in N4, since all the data needs to be read. However, in the other nodes, Avro does not perform that well because of the column pruning applied by Parquet.

The experiments show that our rules work well in realistic scenarios such that of TPC-H. In Figure 10 the overall execution time of a DIW when a single

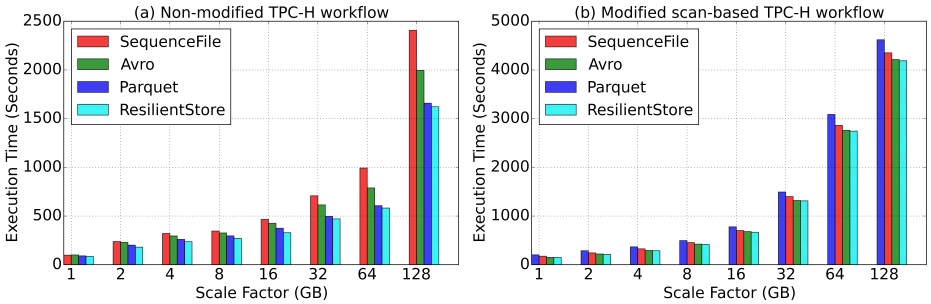

Fig. 10: Single Fixed Format vs ResilientStore

fixed format is chosen is compared against ResilientStore. Figure 10a shows the overall performances in the TPC-H queries. For these queries our approach on average provides 32% speedup over fixed SequenceFile, 19% speedup over fixed Avro, 4% speedup over fixed Parquet and overall, it provides 18% speedup. However, these queries have a very low selectivity factor (i.e., the median is 0.8%) [1], which benefits Parquet. To exemplify a scenario where full-scans would dominate (e.g., for computing data mining algorithms), we modified the TPC-H queries to transform them into scan-based ones (i.e., 100% selectivity factor). In such scenario, ResilientStore chooses Avro for N1 and N2 instead of Parquet. Figure 10b shows the overall performances for the modified queries of TPC-H. In average, our approach provides a 9% speedup over fixed SequenceFile, 1.5% speedup over fixed Avro, 21% speedup over fixed Parquet and overall, it provides 10% speedup. Moreover, this figure also shows that our rules have chosen the right format for all the materialized nodes.

6 Related Work

The fixed format problem has been identified by the research community and many solutions have been proposed. The existing solutions allow using multiple layouts together. For instance, the in-memory DBMS SAP HANA [8] uses horizontal and vertical layouts for On-line Transaction Processing (OLTP) and On-line Analytical Processing (OLAP) workloads, respectively. In a similar way, in DB2 [17] horizontal and vertical layouts can be used for the same table-space. However, these layouts are fixed and non-modifiable at runtime. There are also solutions that consider workloads in order to decide for the most suitable layout. These systems, however, work in multi-database environments. Polybase [5] for instance, is a system that uses both a Hadoop cluster and a DBMS for data storage. Based on the workloads, it dynamically decides which solution is the best. According to this decision, it also moves the data from one system to another for executing queries. This solution focuses on utilizing the processing power of the Hadoop cluster and it always uses an horizontal layout to store data on Hadoop. Similar to Polybase, there is a hybrid system [12], which can read raw files directly and choose the layout based on the input queries. However, they propose to keep multiple copies of the same data in different formats. But this might not be feasible when the size of the data is huge. In addition, there are two systems [7, 18] that store the data inside different storage engines

by taking into account the data access patterns. These systems work like mediators, they analyze the characteristics of the data and then route them to the most suitable storage engine. In [7], the system requires training in order to take the right decision in choosing the best storage engine. This training runs every query in all available systems to see which system fits best each query. Hence, this requires extra processing and adds extra cost. In [18], the solution relies on annotations which are defined by the user during the requirements definition process of an application. These annotations help the mediator to decide where to store the application data. The annotations, however cannot be defined at run-time. Moreover, this solution mainly focuses on choosing a storage engine according to the application requirements without considering the data format.

H2O [2] can dynamically decide the layout of the data based on the current workload. However, it only considers vertical layouts by creating different column groups. As discussed in [2], creating column groups is a NP-hard problem and it is not feasible for tables with many columns. Similarly, Trojan [13] is an adaptable column storage for Hadoop, which takes advantage of the data replication feature of HDFS. It analyzes the workload access patterns and stores different column groups on each replica according to the different access patterns. Then, it routes a query to the most suitable replica format. However, it only considers vertical layouts. Finally, WWHow [14] proposes a data layer which is independent of the physical storage. This layer enables an adaptable physical storage engine by analyzing the application needs. However, they are considering general storage systems such as file-systems, databases and cloud storage without considering the physical formats. Moreover, once decided, the storage system remains fixed.

7 Conclusion and Future Work

Analytical querying introduces variable types of workloads that co-exist in the same system, and a fixed data format does not yield the best performance for all types of workloads. Thus, we discussed the need to introduce flexibility in the data format and decide it based on the characteristics of the subsequent operators accessing data. Specifically, we have shown for the Hadoop ecosystem that selecting the data format according to the access patterns helps in reducing the load time of the intermediate results. In this paper we introduced ResilientStore a tool to assist on selecting the most appropriate data format. ResilientStore analyzes the access patterns of intermediate results and chooses a format by applying heuristic rules. Our experiments on the Hadoop ecosystem show the benefits on performance. In the future work we plan to combine our approach with a cost-based one in a two-phase approach. In the first phase, we use our rule-based approach to choose a format, so we can immediately react to new flows with no overhead. In the second phase, we plan to refine our first decision by gathering the needed statistics (e.g., the operators selectivity factor) and follow an off-line cost-based approach. This way, for future executions of these intermediate results we can refine the rule-based decision made once the needed statistics have been gathered.

Acknowledgments

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. A. Abelló, J. Ferrarons, and O. Romero. Building Cubes with MapReduce. In *DOLAP*, 2011.
2. I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A Hands-free Adaptive Store. In *SIGMOD*, 2014.
3. Y. Chen, S. Alspaugh, and R. Katz. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. In *VLDB*, 2012.
4. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.
5. D. J. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split Query Processing in Polybase. In *SIGMOD*, 2013.
6. I. Elghandour and A. Aboulnaga. ReStore: Reusing Results of MapReduce Jobs. In *VLDB*, 2012.
7. A. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, S. Madden, D. Maier, T. Mattson, S.Papadopoulos, J. Parkhurst, N. Tatbul, M. Vartak, and S. Zdonik. A Demonstration of the BigDAWG Polystore System. In *VLDB*, 2015.
8. F. Färber, S. K. Cha, J. Primsch, C. Bornhovd, S. Sigg, and W. Lehner. SAP HANA Database - Data Management for Modern Business Applications. In *SIGMOD Record*, 2011.
9. A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. Column-Oriented Storage Techniques for MapReduce. In *VLDB*, 2011.
10. S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.
11. Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems. In *ICDE*, 2011.
12. S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *CIDR*, 2011.
13. A. Jindal, J.-A. Quian-Ruiz, and J. Dittrich. Trojan Data Layouts: Right Shoes for a Running Elephant. In *SOCC*, 2011.
14. A. Jindal, J.-A. Quian-Ruiz, and J. Dittrich. WWHow! Freeing Data Storage from Cages. In *CIDR*, 2013.
15. P. Jovanovic, O. Romero, A. Simitsis, and A. Abelló. Incremental Consolidation of Data-Intensive Multi-flows. In *TKDE*, 2016.
16. V. Kalavri, H. Shang, and V. Vlassov. m2r2: A Framework for Results Materialization and Reuse. In *BDSE*, 2013.
17. V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang. DB2 with BLU Acceleration: So Much More than Just a Column Store. In *VLDB*, 2013.
18. M. Schaarschmidt, F. Gessert, and N. Ritter. Towards Automated Polyglot Persistence. In *BTW*, 2015.