Received: 26 September 2018 | Revised: 29 July 2019 | Accepted: 1 August 2019
DOI: 10.1002/cpe.5523

WILEY

RESEARCH ARTICLE

The impact of columnar file formats on SQL-on-hadoop engine performance: A study on ORC and Parquet

Todor Ivanov[1] | Matteo Pergolesi[2]

[1]Frankfurt Big Data Lab, Goethe University Frankfurt, Frankfurt, Germany
[2]Department of Engineering, University of Perugia, Perugia, Italy

Correspondence
Todor Ivanov, Frankfurt Big Data Lab, Goethe University Frankfurt, Frankfurt, Germany.
Email: todor@dbis.cs.uni-frankfurt.de

Summary

Columnar file formats provide an efficient way to store data to be queried by SQL-on-Hadoop engines. Related works consider the performance of processing engine and file format together, which makes it impossible to predict their individual impact. In this work, we propose an alternative approach: by executing each file format on the same processing engine, we compare the different file formats as well as their different parameter settings. We apply our strategy to two processing engines, Hive and SparkSQL, and evaluate the performance of two columnar file formats, ORC and Parquet. We use BigBench (TPCx-BB), a standardized application-level benchmark for Big Data scenarios. Our experiments confirm that the file format selection and its configuration significantly affect the overall performance. We show that ORC generally performs better on Hive, whereas Parquet achieves best performance with SparkSQL. Using ZLIB compression brings up to 60.2% improvement with ORC, while Parquet achieves up to 7% improvement with Snappy. Exceptions are the queries involving text processing, which do not benefit from using any compression.

KEYWORDS

BigBench, big data benchmarking, columnar file formats, Hive, ORC, Parquet, SparkSQL, SQL-on-Hadoop

1 | INTRODUCTION

In the last years, Hadoop has become the standard platform for storing and managing Big Data. However, the lack of skilled developers to write MapReduce programs has pushed the adoption of SQL dialects into the Hadoop Ecosystem in an attempt to benefit from the existing relational database skills, especially in the Business Intelligence and Analytics departments. Apache Hive[1,2] has emerged as the standard data warehouse engine on top of Hadoop. This adoption by the industry has lead the developer community to continuously work on improvements both in query execution path as well as in data storage strategies. For example, Huai et al[3] proposed an effective query planning, a vectorized query execution model, and a new file format, called Optimized Record Columnar (ORC). Similarly, Parquet,[4,5] inspired by the Google Dremel paper,[6] is another columnar file format.

Columnar file formats store structured data in a column-oriented way. The advantages of the columnar over the row oriented formats[3,4] are the following.

- They make it efficient to scan only a subset of columns. If a query accesses only a few columns of a table, the I/O can be drastically reduced compared to traditional row oriented storage where you are forced to read entire rows.
- Organizing data by columns, chunks of data of the same type are stored sequentially. Encoding and compression algorithms can take advantage of the data type knowledge and homogeneity to achieve better efficiency both in terms of speed and file size.

Many SQL-on-Hadoop engines exist nowadays along with file formats designed to accelerate the data access and maximize the storage capacity. Table 1 summarizes the most popular SQL-on-Hadoop engines together with the data file formats they support. As shown in column*Default File Formats*, each engine prefers different default file format with which it achieves its best performance. For example, ORC is favored by Hive[1,2] and

TABLE 1 Popular SQL-on-Hadoop engines

Engines	Default File Format	Other Supported File Formats
Hive[1,2]	ORC	Text, Sequence File, RCFile, Parquet, Avro
Spark SQL[7]	Parquet	Text, JSON, Sequence File, RCFile, ORC, Avro
Impala[8]	Parquet	Text, Sequence File, RCFile, Avro
Pig[9]	Pig Text File	Sequence File, RCFile, ORC, Parquet, Avro
Drill[10]	None	Text, JSON, Sequence File, MapR-DB, Parquet, Avro
Presto[11]	ORC	Text, JSON, Sequence File, RCFile, ORC, Parquet
Tajo[12]	Text File	JSON, Sequence File, RCFile, ORC, Parquet
HAWQ[13]	AO, CO, Parquet	PXF, Text, RCFile, Avro, Hbase
IBM BigSQL[14]	Text File	Sequence File, RCFile, ORC, Parquet, Avro
Phoenix[15]	CSV, JSON	Hbase, Spark RDD and DataFrames
AsterixDB[16]	ADM (super-set of JSON)	ADM, CSV
Vertica[17]	internal raw formats	ORC, Parquet
AWS Athena/Presto[18]	ORC	ORC, Parquet, CSV, JSON, Avro

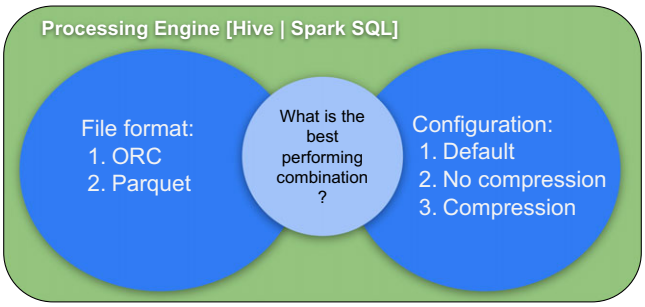

FIGURE 1 Research objective

Presto,[11] whereas Parquet is first choice for SparkSQL[7] and Impala.[8] A number of studies[19-22] have investigated and compared the performance of file formats running them on different SQL-on-Hadoop engines. However, because of the different internal engine architectures, these works actually compare the engine together with its file format optimizations. Contrary to those approaches, in this paper, we compare ORC and Parquet file formats while keeping the processing engine fixed. Our main goal is not to tell which engine is better but to understand how the overall performance of an engine is influenced by a change in the file format type or by a different parameter configuration.

Figure 1 shows a graphical representation of our research objective.

This study investigates the performance of the ORC and Parquet file formats first in Hive and then in Spark SQL. It also shows how tuning accordingly the file format configurations can influence the overall performance. We perform a series of experiments using the standard BigBench (TPCx-BB) benchmark[23] with a dataset size of 1000 GB, comparing different ORC and Parquet configurations.

The contributions of this paper are as follows:

- performance evaluation of ORC and Parquet file formats with their default configuration on both Hive and Spark SQL engines;
- performance comparison of ORC and Parquet file formats with two optimized configurations (respectively with and without data compression) in Hive and Spark SQL;
- investigate the influence of data compression (Snappy) on the file format performance;
- detailed query analysis of representative BigBench queries.

This work is a continuation of a series of benchmark experiments[24-29] conducted at the Frankfurt Big Data Lab.

The rest of this paper is organized as follows. Section 2 gives background information and related work. Section 3 presents the experimental setup and the preparation stages. Section 4 discusses benchmark results for Hive, and Section 5 discusses benchmark results for SparkSQL. Section 6 presents an in-depth analysis of a small subset of representative queries. Section 7 discusses our findings and directions for future work. Finally, Section 8 summarizes our results.

2 | BACKGROUND AND RELATED WORK

In this section, we briefly introduce the main technologies and terms used throughout the paper. Moreover, we present a summary of the most relevant studies investigating data file formats in SQL-on-Hadoop systems.

2.1 | Hive

Apache Hive[1,2] is a data warehouse infrastructure built on top of Hadoop. Hive was originally developed by Facebook and supports the analysis of large data sets stored on HDFS by queries in an SQL-like declarative query language, called *HiveQL*. It does not strictly follow the SQL-92

standard. Additionally, natively calling *User Defined Functions (UDF)* in *HiveQL* allows to filter data by custom Java or Python scripts. Plugging in custom scripts in *HiveQL* makes the implementation of natively unsupported statements possible. Hive consists of two core components: the *driver* and the *Metastore*. The driver is responsible for accepting HiveQL statements, submitted through the command-line interface (CLI) or the HiveServer, and translating them into jobs that are submitted to the MapReduce engine.[2] This allows users to analyze large data sets without actually having to develop MapReduce programs themselves. The *Metastore* is the central repository for Hive's metadata and stores all information about the available databases, tables, table columns, column data types, and more. The *Metastore* uses typically a traditional RDBMS like MySQL to persist the metadata.

2.2 | Spark SQL

Apache Spark[30] is a cluster computing system that is able to run batch and streaming analysis jobs on data distributed on the cluster. Spark SQL[7] is one of the many high level tools running on top of a Spark cluster. It facilitates the processing of structured data by offering an SQL-like interface and support for HiveQL and Hive UDFs. To achieve this, it defines the concept of *DataFrame*, a collection of structured records, and a declarative API to manipulate it. Spark SQL includes a specific query optimizer, *Catalyst*, which improves computation performance thanks to the available information about the data structure. Data can be queried from multiple sources, among which the Hive catalog.

2.3 | Columnar file formats

Apache ORC[3,31] and Apache Parquet[5] are the most popular and widely used file formats for Big Data analytics and they share many common concepts in their internal design and structure. In this section, we will present the main aspects of columnar file formats in general and their purpose in optimizing query execution. Figure 2 will support the explanation.

The upper right part of Figure 2 shows a table example with four columns (A, B, C, and D). Each column has a different color to better understand the column-oriented storage pattern later. A table is stored in one or more HDFS files, composed of one or more file system blocks. Table rows are partitioned into groups and stored in the blocks, as shown in the left upper part of Figure 2. Row groups are data units independent from each other and used to split the input data for parallel processing. Data in row groups is not stored row by row, but column by column. Looking at the first row group detail in Figure 2, we can see that data values for *Column A* (in red) are stored first, next to it come data values for *Column B* (in purple), and so on. These portions of column data are usually called *column chunks* and they allow the filtering of unneeded columns by the query while reading the file. Finally, data in column chunks is split into pages which are the indivisible unit for encoding and compression. Because values in a column chunk share the same data type, encoding algorithms can achieve a more efficient representation. The encoded output is then compressed with a generic compression algorithm (like ZLIB[32] or Snappy[33]).

While the column-oriented storage strategy is used to filter out unnecessary column data for a query, indexes are used to filter out row groups. They are usually placed in front of each row group, so that, by just reading the index, an entire row group can be immediately skipped. This is only possible when the query execution engine uses *predicate push-down* optimization.[34] Indexes are not shown in Figure 2 for the sake of simplicity.

Tables 2 and 3 report all the mentioned general concepts and their specific implementation name for each file format. Furthermore, parameter name and default value for the Hive engine are shown. We do not report parameters for Spark in the tables because our benchmarking platform (BigBench) generates the dataset with Hive and the file format configuration happens in Hive settings.

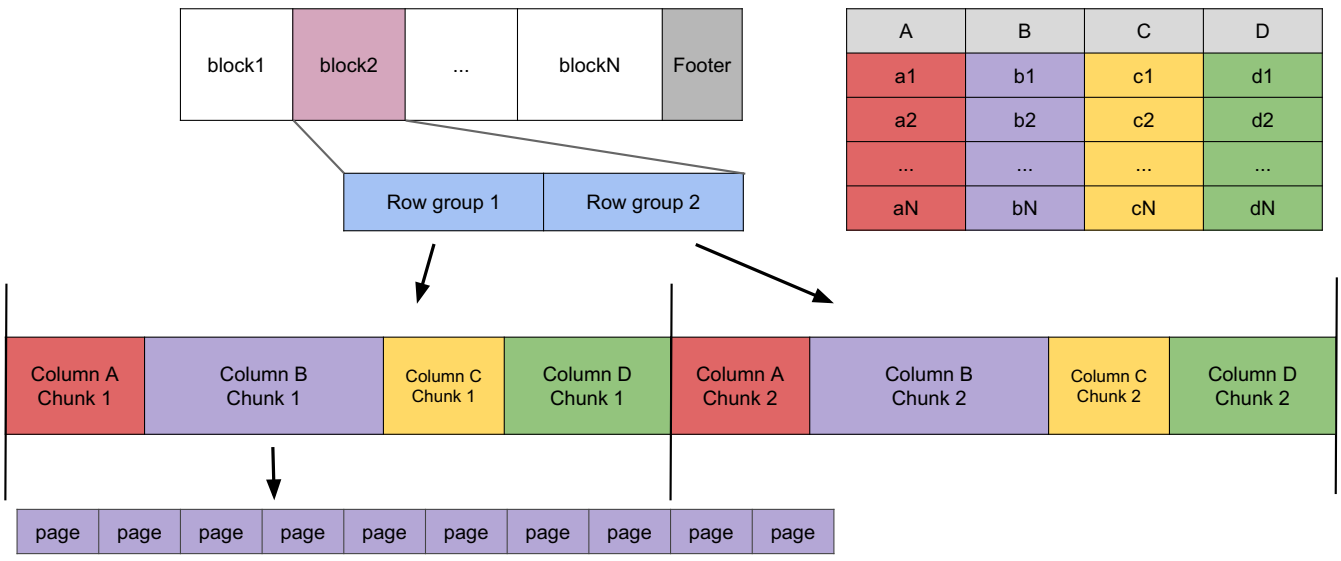

FIGURE 2 General structure of a columnar file

TABLE 2 ORC design concepts and default configuration

Concept	Name	Hive Configuration	Default
Group of rows	Stripe	hive.exec.orc.default.stripe.size	67 108 864 Bytes
Index	Index	orc.create.index (not available in Hive)	true
		hive.exec.orc.default.row.index.stride	10 000 rows
Portion of column	Stream		
Page	Compression chunk	hive.exec.orc.default.buffer.size	262 144 Bytes
Encoding	Encoding	hive.exec.orc.encoding.strategy	SPEED
Compression	Compression	hive.exec.orc.default.compress	ZLIB

TABLE 3 Parquet design concepts and default configuration

Concept	Name	Hive Configuration	Default
Group of rows	Row group	parquet.block.size	134 217 728 Bytes
Index	Dictionary page	parquet.enable.dictionary	true
		parquet.dictionary.page.size	1 048 576 Bytes
Portion of column	Column chunk		
Page	Data page	parquet.page.size	1 048 576 Bytes
Encoding	Encoding	parquet.enable.dictionary	True
Compression	Compression	parquet.compression	Uncompressed

2.4 | Optimized record columnar file

Apache optimized record columnar (ORC)[3,31] is a self-describing (includes metadata), type-aware columnar file format designed initially for Hadoop workloads, but now used as a general purpose storage format. It is optimized for large streaming reads and has many advantages over its predecessor, the RCfile format.[35]

Table 2 shows the ORC implementation of general design concepts we talked about in Section 2.3. A group of rows is called a *stripe* in ORC. The size is configurable and defaults to 64MB. Stripes are explicitly separated from each other by an index (placed in front) and a footer. A stripe contains portions of the table columns, which are called *streams* and each stream is divided into pages, called *compression chunks*. Pages are sized 256 KB by default.

The index contains row statistics and the minimum and maximum value for each column stream. An index can be stored into more pages and a page contains information about 10 000 rows by default. In addition, a bloom filter is included for better row filtering. Finally, a top level index is placed in the file footer.

ORC supports the complete set of data types available in Hive, including the complex types: structs, lists, maps, and unions. Numerical columns are encoded using Run-Length Encoding (RLE) and it is possible to select between the *SPEED* (default) or *COMPRESSION* strategy. Dictionary encoding is applied when possible to strings. The latter makes the encoding more lightweight; anyway, this has nothing to do with the generic compression algorithm, which is applied on the encoding output. Default compression algorithm is ZLIB.[32]

The metadata in ORC is stored at the end of file (after the file footer) using the Protocol Buffers,[36] providing the ability to add new fields to the table schema without breaking readers.

2.5 | Parquet

Apache Parquet[5] is an open source columnar storage format using complex nested data structures inspired by the Google Dremel paper.[6] It is a general purpose storage format that can be used or integrated with any data processing framework or engine. Parquet supports efficient compression and encoding schemas on a per-column level. It uses Apache Thrift[37] for the metadata definitions. There are three types of metadata: file metadata, column (chunk) metadata, and page header metadata.

Table 3 shows the Parquet implementation of general design concepts we talked about in Section 2.3. The row group concept keeps the same name in Parquet documentation, whereas it is called *block* in Hive configuration and has a default size of 128 MB. Differently from ORC, row groups are not explicitly separated from each other. Furthermore, for column chunks, Parquet keeps using the general name. Pages composing a column chunk are called *data pages* to distinguish them from *dictionary pages* used as indexes. The default size of each data page is 1 MB, 4 times larger than ORC compression chunks.

Column chunk data is paired with metadata that includes a *dictionary page*, a compact representation of column values. The dictionary pages are useful to filter out unnecessary data for the query. Dictionary page size is customizable and it defaults to 1 MB, the same default value used for data pages.

At the end of the file, metadata describing the file structure are stored. File metadata contain references to all of the column chunk metadata start locations to easily access them. Furthermore, it allows to immediately filter out columns not needed by the query.

For encoding, Parquet uses a dictionary encoding when applicable on text data. RLE or bit-packing are used for numerical values. The selection happens automatically when writing data to a Parquet file. A further generic compression algorithm can be applied on encoded data. In the default Hive configuration, the Parquet data is not compressed.

2.6 | Related work

In recent years, multiple studies evaluated and compared the different SQL-on-Hadoop engines along with the file formats they support. A recent SQL-on-Hadoop Tutorial[38] at VLDB 2015 reviews in detail the architectures of the most popular ones. ORC and Parquet are also listed as the most widely used file formats.

Performance Comparisons. Table 4 summarizes related work that evaluate SQL-on-Hadoop engines with ORC and Parquet.

Floratou et al[19] compared the performance of ORC and Hive with the one of Parquet and Impala using TPC-H[42] and TPC-DS[43] queries. The results show that Impala is 3.3× to 4.4× faster than Hive on MapReduce and 2.1× to 2.8× faster than Hive on Tez for the TPC-H experiments. For the TPC-DS inspired experiments, Impala 8.2× to 10× faster than Hive on MapReduce and about 4.3× faster than Hive on Tez. The results also show that Parquet skips data more efficiently than the ORC format, which tends to prefetch unnecessary data especially when a table contains a large number of columns. However, the built-in index in ORC format mitigates that problem when data is sorted.

Similarly, Chen et al[21] compare multiple SQL-on-Hadoop engines using modified TPC-DS queries on clusters with varying number of nodes. In terms of storage formats, they use the default ORC and Parquet configuration parameters. The results show that overall Impala and Shark are the fastest followed by Presto, Hive, and Stinger. Moreover, Shark and Impala perform better on small datasets, and Hive, Stinger, and Shark are sensitive to data skewness.

Another work by Wouw et al[20] presented a new benchmark with real and synthetic data and compared Shark, Impala, and Hive in terms of processing power, resource utilization, and scalability. The results do not show a clear winner in terms of performance, but Impala and Shark have similar behavior. In terms or resource consumption, Impala is the most CPU efficient and has slightly less disk I/O than Shark and Hive.

Costea et al[22] introduced the VectorH engine as a new SQL-on-Hadoop system on top of Vectorwise and compared it with similar engines using TPC-H. In the experiments, they used ORC and Parquet with Snappy compression. The experiments showed that VectorH performs 1-3 orders of magnitude faster than Impala (with Parquet), Hive (with ORC), HAWQ (with Parquet), and Spark SQL (with Parquet), thanks to the multiple optimizations introduced in the paper.

A recent work by Poggi et al[39] evaluated the Hive on MapReduce and Hive on Tez (with default ORC format configuration) performance on multiple cloud providers using the TPC-H benchmark. The results showed that the price-to-performance ratio for the best cloud configuration is within a 30% cost difference for the 1TB scale. The same team[40] evaluated different cloud providers and their Hive+Tez/Hive+MR as well as Spark offerings using the BigBench (TPCx-BB) benchmark. All experiments were performed using data stored in ORC format and default configurations unless there were specific execution problems. The results showed that Hive-on-Tez performs up to 4x better than Hive-on-MapReduce. Hive-on-Tez is also faster than Spark 2.1 on the lower scale factors, but this difference narrows down for the larger data sizes. However, it is not possible to draw any conclusions if ORC or Parquet influenced the overall performance.

Last but not the least, Pirzadeh et al compared different SQL-on-Hadoop engines (Hive, Spark SQL, AsterixDB, and a commercial parallel relational database) stressing them with TPC-H and storing the underlying data in various file formats. Similar to other studies, the authors

TABLE 4 Summary of related work

Citation	Engines	Benchmark	File Formats	Configurations
Floratou et al[19]	Hive v0.12 Hive-Tez v0.13 Impala v1.2.2	TPC-H TPC-DS	ORC Parquet	Text Default Snappy
Chen et al[21]	Hive v0.10 HiveStinger v0.12 Shark v0.7.0 Impala v1.0.1 Presto v0.54	TPC-DS	ORC Parquet	Text Default
Wouw et al[20]	Hive v0.12 Impala v1.2.3 Shark v0.8.1	CALDA real world dataset	Sequence file	Snappy
Costea et al[22]	Hive v1.2.1 VectorH v5.0 Impala v2.3 HAWQ v1.3.1 Spark SQL v1.5.2	TPC-H	ORC Parquet VectorH	ORC+Snappy (Hive) Parquet+Snappy (Impala, HAWQ, SparkSQL) VectorH+LZ4
Poggi et al[39]	Hive v1.2.1 Hive+Tez v1.2.1	TPC-H	ORC	Default
Poggi et al[40]	Hive+Tez v1.2-2.1 Spark+SQL+MLlib v1.6-2.1	BigBench (TPCx-BB)	ORC	Default
Pouria et al[41]	Hive+Tez v1.2 AsterixDB v0.8.9 Spark v1.5	TPC-H	ORC (Hive) Parquet (SparkSQL)	Default

compared SparkSQL with text data and Parquet; Hive-on-MR and Hive-on-Tez with ORC; and AsterixDB with normalized data and nested data. The results showed that using optimized columnar file formats such as ORC and Parquet significantly improved the performance.

Optimizations. Recently, many new techniques for optimizing the performance and efficiency of analytical workloads on top of columnar storage formats have been proposed. Bian et al[44] focused on improving the data scan throughput by finding an optimal column layout. By applying efficient column ordering, they reduced the end-to-end query execution time by up to 70% and respectively by additional 12% when using column duplication with less than 5% extra storage.

A different approach of reducing data access through data skipping was presented by Sun et al.[45,46] The authors introduced a four-step framework (workload analysis, augmentation, reduce, and partitioning phases) for data skipping by applying more effective partitioning schema that takes into account the query filters. The results showed 3-7x improvements in the query response time compared to the traditional range partitioning. In their latest work, Sun et al[47] presented a novel hybrid data skipping framework that optimizes the overall query performance by automatically balancing skipping effectiveness and tuple-reconstruction overhead. It allows both horizontal and vertical partitioning of the data, which maximizes the overall query performance.

At the same time, new file formats utilizing the capabilities of emerging storage components like Non-Volatile Memory (NVMe) devices were introduced. Apache Arrow[48] provides a language-independent columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware. Trivedi et al[49] introduced a new high-performance file format optimized for NVMe devices that achieves up to 21.4x performance gains. The authors integrated it with SparkSQL and showed up to 3x query accelerations with TPC-DS.

However, none of the above studies investigates the similarities and differences of both formats using a common SQL-on-Hadoop engine as a baseline. Additionally, most of the above comparisons were based on benchmarks using structured data such as the TPC-H[42] and TPC-DS.[43] In our study, we use BigBench, which operates on structured, semi-structured, and unstructured data and has been standardized as TPCx-BB[50] by the TPC committee.

3 | EXPERIMENTAL SETUP

This section describes the hardware and software components and the different configurations used in our experiments.

3.1 | Hardware configuration

The experiments were performed on a cluster consisting of 4 nodes connected directly through a 1GBit Netgear switch. All 4 nodes are Dell PowerEdge T420 servers. The master node is equipped with 2x Intel Xeon E5-2420 (1.9 GHz) CPUs each with 6 cores, 32 GB of main memory, and 1 TB hard drive. The 3 worker nodes are equipped with 1x Intel Xeon E5-2420 (2.20 GHz) CPU with 6 cores, 32 GB of RAM, and 4x 1 TB (SATA, 7.2K RPM, 64 MB Cache) hard drives.

3.2 | Software configuration

All four nodes in the cluster are equipped with Ubuntu Server 14.04.1 LTS as the operating system. On top of that, we installed the Cloudera Distribution of Hadoop (CDH) version 5.11.0, which provides Hadoop, HDFS, and Yarn all at version 2.6.0 and Hive 1.1.0. Separately, we installed Spark 2.3.0 and we configured Spark SQL to work with Yarn and Hive catalog. The total storage capacity is 13 TB of which 8 TB are effectively available as HDFS space. Due to the resource limitation (only 3 worker nodes) of our setup, the cluster was configured to work with replication factor of two. Table 5 show relevant cluster parameters and how they were adjusted for the experiments.

TABLE 5 Cluster configuration

Component	Parameter	Configuration Value
YARN	yarn.nodemanager.resource.memory-mb	31 GB
	yarn.scheduler.maximum-allocation-mb	31 GB
	yarn.nodemanager.resource.cpu-vcores	11
Spark	master	yarn
	num-executors	9
	executor-cores	3
	executor-memory	9 GB
	spark.serializer	org.apache.spark.serializer.KryoSerializer
MapReduce	mapreduce.map.java.opts.max.heap	2 GB
	mapreduce.reduce.java.opts.max.heap	2 GB
	mapreduce.map.memory.mb	3 GB
	mapreduce.reduce.memory.mb	3 GB
Hive	Client Java Heap Size	2 GB

3.3 | BigBench

In order to perform an extensive evaluation of the file formats, it was necessary to use a Big Data benchmark that is able to evaluate the 3Vs characteristics and utilizes structured, semi-structured, and unstructured data. BigBench[23,51] has been proposed and developed to address exactly these needs. It is an end-to-end analytics, application level, and technology agnostic Big Data benchmark that was recently adopted by TPC and released as the TPCx-BB[50] benchmark. Chowdhury et al[52] presented a BigBench implementation for the Hadoop ecosystem, which is available on GitHub[53] and was used for our experiments. The data set is generated on a fictitious product retailer business model, whereas the workload consists of 30 complex queries. 10 queries were taken from the TPC-DS benchmark,[43] whereas the remaining 20 queries were adapted from the McKinsey report.[54] Table 6 summarizes the number and type of queries of this BigBench implementation. The query grouping is an extended version of the one proposed in BigBench specification (see appendix B.3.2 in the work "TPC - Current Specifications - tpcx-bb v1.2.0"[55]). All queries are implemented using Apache Hadoop,[56] Hive,[57] Mahout,[58] and the open Natural Language Processing toolkit.[59] Apache Mahout[58] is a library for quickly creating scalable performing machine learning applications on top of MapReduce, Spark, and similar frameworks. Apache OpenNLP[59] is a library toolkit for machine learning processing of natural language text.

3.4 | File format configurations

One of the goals of this study is to investigate how changing the configuration parameters of ORC and Parquet influence their performance. Therefore, defining the exact configuration parameter values was the first very essential step before starting the experiments.

As shown in Sections 2.3, 2.4, and 2.5, the two file formats share many concepts in their structure design. Our goal is to setup ORC and Parquet with a similar configuration, so to be able to meaningfully compare their performance. We define three test configurations reported in Table 7 and we focus on three parameters: *the row group size*, *the page size*, and *the compression algorithm*. All other parameters are set to their default values for all the three test configurations. As expected, the use of indexes in ORC and of dictionary pages in Parquet is enabled by default. We set the HDFS block size to 256 MB for all of our tests.

The first test configuration is called *Default Config* and uses the default ORC and Parquet parameters as stated in each file format documentation. The two formats have very different configurations, especially in the compression parameter. ORC uses ZLIB compression and Parquet does not use any compression, which makes the benchmark results not comparable. Nevertheless, we decided to keep this setup to show the file formats behavior when using them "out-of-the-box" and to highlight the performance change after an optimized parameter configuration.

The two other configurations, named *Snappy Config* and *No Compression Config*, use respectively the same value for row group and page size while the compression algorithm changes. We increase the row group size to 256 MB to have more sequential reads from disk at the expense of a higher memory usage.[3,44] The page size is set to 1 MB (default for Parquet). A larger page size improves the compression performance and decreases overhead, again, at the expense of a higher memory usage.

The compression parameter is also very important as it determines which general-purpose algorithm such as Snappy, ZLIB, or LZO is used after the file format encoding.[3] Using compression, file readers perform less I/O operations to get the data from disk but more CPU cycles are spent to decompress them. In our configurations, we use the *Snappy* compression, which is supported by both ORC and Parquet.

Many parameter combinations are possible and would lead to interesting research questions. For example, how the performance for the two file formats change if we vary the row group and page size from 64 MB to 256 MB in steps of 16 MB? Due to time constraints, we limit our tests to these three configurations to achieve a meaningful comparison of the two file formats when running queries on a fixed engine.

TABLE 6 BigBench query types

Query Types	Queries	Number of queries
Pure HiveQL	6,7,9,11,12,13,14, 15,16,17,21,22,23,24	14
MapReduce /UDTF	1	1
MapReduce /Python	2,3,4,8,29,30	6
HiveQL /Spark MLlib	5,20,25,26,28	5
MapReduce /OpenNLP	10,18,19,27	4

TABLE 7 File Format Configurations

Format	Parameters	Default config	No compression config	Snappy config
Parquet	parquet.block.size	128 MB	256 MB	256 MB
	parquet.page.size	1 MB	1 MB	1 MB
	parquet.compression	uncompressed	uncompressed	snappy
ORC	hive.exec.orc.default.stripe.size	64 MB	256 MB	256 MB
	hive.exec.orc.default.buffer.size	256 KB	1 MB	1 MB
	hive.exec.orc.default.compress	zlib	uncompressed	snappy

3.5 | Engine configuration

To fully take advantage of columnar file formats design, it is important to configure some settings in the query execution engine. It is important to notice that these parameters do not affect the file generation, but only the query processing.

Code 1 SparkSQL query execution settings.

```
1  spark.sql.parquet.filterPushdown                    true
2  spark.sql.parquet.recordLevelFilter.enabled         true
3  spark.sql.hive.convertMetastoreParquet              true
4  spark.sql.orc.filterPushdown                        true
```

For SparkSQL, we edit spark-default.conf file by adding the lines shown above.[60,61] The first group of parameters is relevant for Parquet file format. Lines 1 and 2 enable full support for predicate push-down optimizations. Parameter at line 3 enables the usage of Parquet built-in reader and writer for Hive tables, instead of SerDe.[62] Finally, line 4 in the previous file snippet enables predicate push-down also for ORC.

Code 2 Hive query execution settings.

```
1  set hive.optimize.ppd=true;
2  set hive.optimize.ppd.storage=true;
3  set hive.ppd.recognizetransivity=false;
4  set hive.optimize.index.filter=true;
```

BigBench uses custom settings for Hive, overriding the default.[63] We set the parameters shown above in engineSettings.sql file placed in BigBench Hive subfolder. Lines 1-3 enable the predicate push-down optimization. Line 4 enables the use of format specific indexes for both ORC and Parquet.

3.6 | Load times and data sizes

The last step before starting the experiments is to generate data using the BigBench data generator based on the PDGF.[64] It generates data in text format that needs to be loaded and converted to a specific file format. The loading process can greatly vary in time depending on the compression used by both ORC and Parquet. BigBench relies only on Hive for creating the schema in the Metastore and storing the file formats data into HDFS. Figure 3 displays the loading times (in minutes) versus the dataset sizes (in GB) for all configurations in Hive version 1.1.0 with scale factor 1000 (1 TB). However, the size of the generated data even with *No Compression Config.* is much smaller (around 600 GB) than 1 TB due to the columnar file format optimizations applied on the stored data (Section 2.3). The detailed summary of all results is available on GitHub[65] under file *Loading-times.xlsx*.

To better compare measurements in Figure 3, we divide it in four equal quadrants by drawing a line at 350 GB on the Y-axis and 150 minutes on the X-axis. There is an obvious trade-off between time taken to generate the data and the size of the generated data. The points in the down-left quadrant will be the one with optimal performance, which in our case is empty. Therefore, the ORC *Default* configuration (using ZLIB compression) in the down-right quadrant has the best performance in terms of time and data size, followed by the ORC *Snappy* configuration with the fastest time generation. Parquet achieves its best performance with the *Snappy* configuration also in the down-right quadrant.

3.7 | Performance evaluation

Our plan is to run experiments with all the 30 BigBench queries with the scale factor 1000 of BigBench (1000 GB of data). All tests are repeated three times and we measure the execution time of each query. The averaged values are taken as a representative number. In order to compare the three different configurations defined in Table 7, we need to repeat the three runs for each configuration for both ORC and Parquet. To better compare the file formats and understand the different support on multiple processing engines, we perform the experiments on two

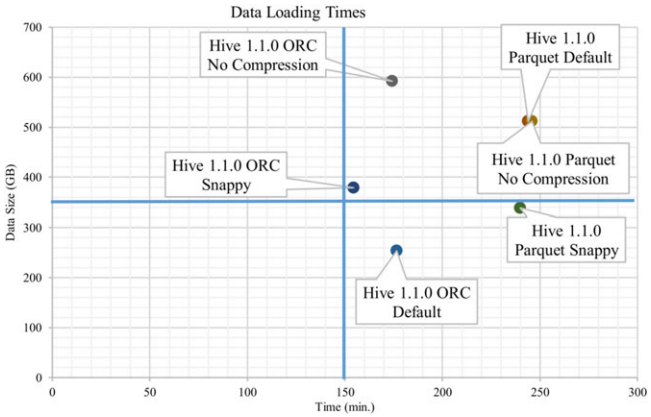

FIGURE 3 Hive Load Times (min.) on the X-axis and Data Sizes (GB) on the Y-axis for Scale Factor 1000

TABLE 8 Experimental Roadmap

Configuration	Default		No compression		Snappy compression	
File Format	ORC	Parquet	ORC	Parquet	ORC	Parquet
BigBench Queries	30		30		30	
Average Total Execution Time per Run (hours) on Hive	24	26	24	26	24	25
Average Total Execution Time per Run (hours) on SparkSQL	12	15	15	14	15	14
Number of Runs	3		3		3	
Total Execution Time (hours)	108	124	116	120	116	117

popular engines, Hive and Spark SQL. Table 8 summarizes all experimental runs to give an idea about the time overhead for our study. The *Average Total Execution Time per Run* is the time needed to run all the 30 BigBench queries. We report this time in hours for Hive and Spark SQL. As stated before, the tests are repeated three times for each combination of file format type, configuration, and processing engine. By summing and multiplying, we obtain the *Total Execution Time* (in hours) needed for each of our configurations. It is reported on the last line of Table 8. By summing all total execution times for the experiments, we get around 701 hours (29.2 days) of testing time.

We defined two metrics to help us compare the different file format configurations. The first metric is called *Performance Improvement (PI%)* and the second one is called *Compression Improvement (CI%)*. When comparing two execution times, we define the higher time *HT* (worst) as the baseline and we compute the time difference with the lower time *LT* (better) as a percentage. This is the amount of saved time between the two executions and we define it as *Performance Improvement (PI%)*:

$$PI\% = \left| \frac{(LT - HT)}{HT} \cdot 100 \right|. \tag{1}$$

Similarly, we define the *Compression Improvement (CI%)* as follows:

$$CI\% = \frac{NoCompressionTime \cdot 100}{CompressionTime} - 100. \tag{2}$$

The *Performance Improvement (PI%)* metric reports with how many % a file format configuration is faster on ORC compared to Parquet or vice verse. It compares the execution times of the different file formats, whereas *Compression Improvement (CI%)* compares the improvement on a particular file format (Parquet or ORC) when using data compression. The *CI%* can be negative when the baseline configuration without any compression performs faster than the configurations with compressions (eg, Snappy and ZLIB).

4 | HIVE

In the next subsections, we show results for the Hive processing engine. Each table is dedicated to a query set identified by the query type as described in Table 6. Note that we merged the single MapReduce/UDTF query with MapReduce/Python group. The first column reports the query number, whereas the following columns report execution times for each combination of configuration (Table 7) and file format type. The *Total Time* of query execution is shown at the end of the table, whereas the very last two rows report the *PI%* and *CI%*. The latter is used to show the overall improvement achieved by using data compression, whereas *PI%* shows the overall performance difference between the two file formats for each configuration.

Query execution time is reported in minutes, and green cells highlight the best time between file formats within each of our configurations (ie, *Default*, *No compression*, *Snappy*). Pairs of cells with the same color (white or light gray) are used to show similar performance between the two file formats: **we consider query performance comparable/equal if the difference is lower or equal to 1 minute.** The standard deviation in % between the 3 query executions for all Hive queries is under 5%, which indicates that Hive achieves very stable execution behavior.

Furthermore, we discuss the results from the perspective of a practitioner who runs BigBench as a black-box benchmark and analyzes the results. This is also the motivation to include the *Default* file format configurations in our tests. We want to show what kind of performance you get by using file formats out-of-the-box and how adjusting the configuration parameters can influence the system behavior.

4.1 | Pure HiveQL

Table 9 shows results for the *Pure HiveQL* query type. Figure 4 depicts the query execution times as visual chart. These queries are fully implemented in HiveQL and, unlike other query types, do not use UDF (User Defined Functions) or external libraries.

Looking at the *Default* configuration column, we can observe a clear pattern with ORC generally performing better. Only Q15, Q16, and Q22 show similar performance. For a naive user, this can be considered as an expected behavior since, as shown in Table 1, ORC is the default file format in Hive and we expect better optimization with this engine. However, the file formats are configured very differently with ORC using

TABLE 9 Hive Results for *Pure HiveQL* query type. Query execution time is reported in minutes. Green cells, as well as <> symbols, highlight the best time between file formats within each configuration. Pairs of cells with similar performance are filled with the same color (white or light gray) and divided by the ≈ symbol. *PI*% and *CI*% respectively show the performance difference between formats and between uncompressed and compressed configurations

	Default (min.)			No compression (min.)			Snappy (min.)		
Query	**ORC**		**Parquet**	**ORC**		**Parquet**	**ORC**		**Parquet**
Q06	40	<	44	43	≈	44	41	<	43
Q07	30	<	34	32	<	34	30	<	34
Q09	16	<	20	16	<	21	16	≈	16
Q11	12	<	14	14	≈	14	13	≈	14
Q12	26	<	35	29	<	35	27	<	30
Q13	27	<	33	30	<	33	28	<	31
Q14	6	<	9	7	<	9	5	<	9
Q15	4	≈	5	4	≈	5	4	≈	4
Q16	62	≈	62	62	≈	62	62	≈	61
Q17	15	<	18	15	<	18	14	<	17
Q21	20	<	25	21	<	25	20	<	24
Q22	28	≈	28	28	≈	28	31	>	28
Q23	8	<	12	9	<	13	8	<	11
Q24	20	<	26	23	<	26	21	<	25
Total Time (min.)	312	<	366	332	<	367	320	<	347
Performance Improvement (PI) %	14.8 %		⟵	9.5 %		⟵	7.8 %		⟵
Compression Improvement (CI) %	6.3 %		baseline	baseline		baseline	3.7 %		5.6 %

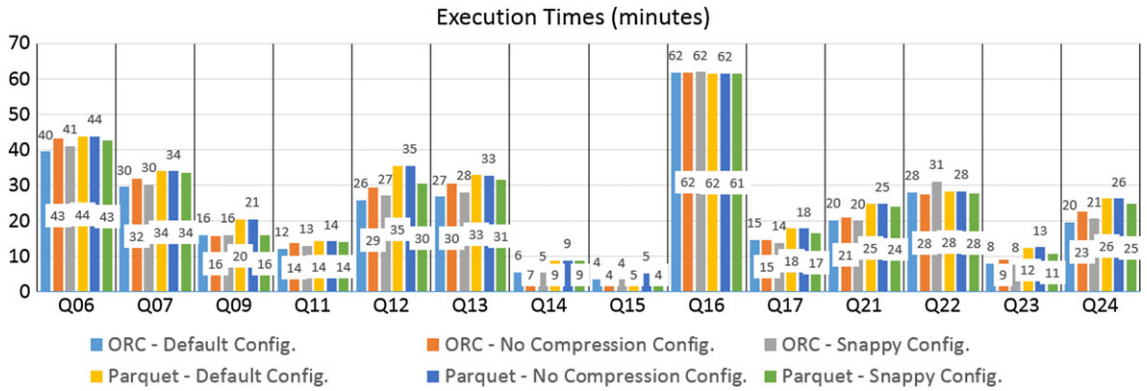

FIGURE 4 Hive Results (in minutes) for the Pure HiveQL queries with all three file format configurations

ZLIB compression and a bigger block size, whereas Parquet is not using compression at all (Table 7). This comparison can then lead to misleading conclusions.

To verify the previous results, we can observe the *No compression* and *Snappy* columns. In these two configurations, we tried to set ORC and Parquet parameters to be as similar as possible, so that the performance comparison is more accurate. Observing the *No compression* column, we can see that the previous pattern with ORC as a winner is confirmed, although the performance difference between the two file formats is reduced. As shown in Table 9, from a ~14.8% PI on *Default* configuration, we move to a ~9.5% PI. Q06 and Q11 are added to the group of queries showing similar performance.

Moving to the *Snappy* column, we can observe a general performance improvement for both file formats. This is shown by comparing the *Total Time* cells. Blue cells show the baseline and CI for ORC (~3.7%), whereas orange cells do the same for Parquet (~5.6% CI). When using compression, the better performance of ORC is again confirmed. Still, Q09 shows better performance on Parquet with similar performance to ORC, and Q22 seems to get worse performance on ORC when adding Snappy compression to the configuration.

Finally, we can state that ORC is the winning file format for the *Pure HiveQL* query type on the Hive engine. Furthermore, using Snappy data compression slightly improves the performance of both file formats.

4.2 | MapReduce/Python

Table 10 shows results for the *MapReduce/Python* query type. Figure 5 depicts the query execution times as visual chart. The queries are implemented in HiveQL and enriched with Python programs to execute complex operations within the query definition.

Starting from the *Default* column, we can observe again that ORC is performing better than Parquet for all the queries. Only Q29 shows a difference of two minutes, but similar performance on the other configurations. We believe that this small difference is only due to noise in

TABLE 10 Hive Results for *MapReduce/Python* query type. Query execution time is reported in minutes. Green cells, as well as <> symbols, highlight the best time between file formats within each configuration. Pairs of cells with similar performance are filled with the same color (white or light gray) and divided by the ≈ symbol. *PI%* and *CI%* respectively show the performance difference between formats and between uncompressed and compressed configurations

	Default (min.)			No compression (min.)			Snappy (min.)		
Query	ORC		Parquet	ORC		Parquet	ORC		Parquet
Q01	12	<	18	14	<	17	13	<	16
Q02	170	<	183	172	<	184	171	<	180
Q03	76	<	91	79	<	90	76	<	85
Q04	155	<	171	156	<	172	155	<	167
Q08	41	<	49	44	<	49	42	<	45
Q29	38	<	40	40	≈	40	39	≈	40
Q30	250	<	259	250	<	258	249	<	253
Total Time (min.)	743	<	811	755	<	811	746	<	786
Performance Improvement (PI) %	8.4 %		⟵	6.8 %		⟵	5.1 %		⟵
Compression Improvement (CI) %	1.7 %		baseline	baseline		baseline	1.3 %		3.1 %

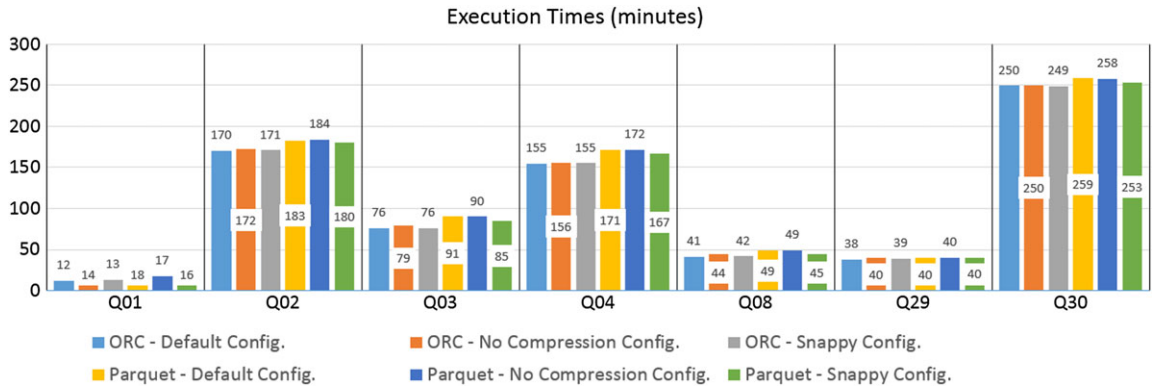

FIGURE 5 Hive Results (in minutes) for the MapReduce/Python queries with all three file format configurations

the execution time and we conclude that Q29 acts similarly in spite of any change in the file format configuration. The table shows a ~8.4% PI between ORC and Parquet with the *Default* configuration.

The performance behavior is confirmed on the *No compression* and *Snappy* columns, where the PI is reduced respectively to ~6.8% and ~5.1%. Both file formats benefit from a small performance improvement thanks to the introduction of compression. This is highlighted in the blue and orange cells at the end of Table 10.

4.3 | HiveQL/OpenNLP

Table 11 shows results for the *HiveQL/OpenNLP* query type. Figure 6 depicts the query execution times as visual chart. The queries are implemented in HiveQL and enriched with a Java UDF to process natural language data (ie, product reviews).

TABLE 11 Hive Results for *HiveQL/OpenNLP* query type. Query execution time is reported in minutes. Green cells, as well as <> symbols, highlight the best time between file formats within each configuration. Pairs of cells with similar performance are filled with the same color (white or light gray) and divided by the ≈ symbol. *PI%* and *CI%* respectively show the performance difference between formats and between uncompressed and compressed configurations

	Default (min.)			No compression (min.)			Snappy (min.)		
Query	ORC		Parquet	ORC		Parquet	ORC		Parquet
Q10	33	>	18	23	>	19	30	≈	30
Q18	50	>	45	46	≈	47	48	<	50
Q19	13	>	9	10	≈	9	12	≈	12
Q27	2	≈	1	2	≈	1	2	≈	2
Total Time (min.)	97	>	73	81		77	92		94
Performance Improvement (PI) %	⟶		25.5 %	⟶		5.1 %	2.3 %		⟵
Compression Improvement (CI) %	-17.3 %		baseline	baseline		baseline	-12.0 %		-18.4 %

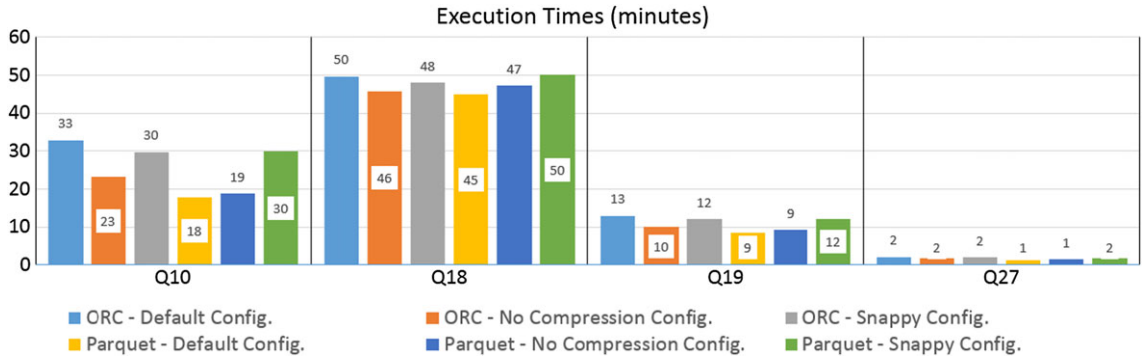

FIGURE 6 Hive Results (in minutes) for the HiveQL/OpenNLP queries with all three file format configurations

Looking at the *Default* column, we observe an opposite performance pattern with respect to the previous query types. Parquet shows considerably better performance than ORC for Q10, Q18, and Q19. Q27 takes a very short time, making it hard to spot any significant performance difference across the various configurations. The PI for *Default* configuration is ~25.5%.

To understand this unexpected behavior, it is useful to look at query performance when the two file formats have a similar configuration. With the *No compression* configuration, the PI resulting from the usage of Parquet decreases to ~5.1%. All the queries show similar performance, with the exception of Q10 that takes a considerably shorter time on Parquet. The results reported in the *Snappy* column highlight a very similar performance behavior between the two file formats. Q18 seems to perform better on ORC here, but with a very small difference of about two minutes. We can conclude that the performance is not greatly influenced by the file format because, if configured similarly, ORC and Parquet show a comparable behavior.

It is interesting to compare the performance between *No compression* and *Snappy*. For the *OpenNLP* query type, the introduction of the Snappy data compression does not bring any benefit, but instead, it worsens the performance. As reported in the very last line of Table 11, both ORC (blue cells) and Parquet (orange cells) show a negative and consistent CI. This observation finally explains the unexpected performance pattern reported in the *Default* column. In fact, Parquet uses no compression as a default, resulting in better performance when compared to ORC that uses ZLIB in its *Default* configuration.

4.4 | HiveQL/Spark MLlib

Table 12 shows results for the *HiveQL/Spark MLlib* query type. Figure 7 depicts the query execution times as visual chart. HiveQL code is used to extract and prepare input data for machine learning algorithms. Usually, input is stored in a temporary table, processed with Spark MLlib Java library, and the output is stored in a new additional table.

Looking at the *Default* column, we can see that all the queries except Q28 perform better on ORC than Parquet. Q28 performance is not affected by any change in the file format configuration. The PI between the two file formats is 8.1%.

Moving to the *No compression* column, the previous performance pattern is confirmed but the PI decreases to ~4.9%. The introduction of Snappy data compression improves the performance for both file formats in a similar amount, respectively ~2.6% for ORC (blue cells) and ~2.5% for Parquet (orange cells). Indeed, ORC results as the file format winner also for the *Snappy* configuration.

TABLE 12 Hive Results for *HiveQL/Spark MLlib* query type. Query execution time is reported in minutes. Green cells, as well as <> symbols, highlight the best time between file formats within each configuration. Pairs of cells with similar performance are filled with the same color (white or light gray) and divided by the ≈ symbol. *PI%* and *CI%* respectively show the performance difference between formats and between uncompressed and compressed configurations

	Default (min.)			No compression (min.)			Snappy (min.)		
Query	**ORC**		**Parquet**	**ORC**		**Parquet**	**ORC**		**Parquet**
Q05	147	<	153	148	<	154	146	<	149
Q20	65	<	69	68	≈	69	66	<	69
Q25	34	<	42	38	<	42	35	<	41
Q26	24	<	30	26	<	30	25	<	28
Q28	6	≈	6	6	≈	6	6	≈	6
Total Time (min.)	276	<	300	286	<	301	279	<	294
Performance Improvement (PI) %	8.1 %		⟵	4.9 %		⟵	5.0 %		⟵
Compression Improvement (CI) %	3.8 %		baseline	baseline		baseline	2.6 %		2.5 %

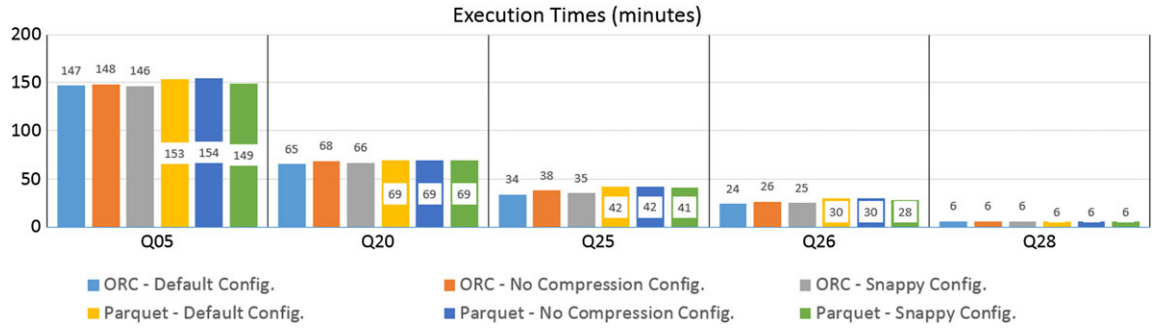

FIGURE 7 Hive Results (in minutes) for the HiveQL/Spark MLlib queries with all three file format configurations

4.5 | Summary

Next, we summarize the major findings of all experiments reported in this section.

1. Using the "out-of-the-box" *Default* file format configuration is not always the optimal choice and depends heavily on the data structure and query type (Section 4.3).
2. ORC generally achieves best performance with the Hive engine with a *Performance Improvement (PI%)* going from ~5% to ~10%. The *HiveQL/OpenNLP* query type (Section 4.3) makes an exception, with Parquet performing better on *No compression* (~ 4.9% of *PI*) and similarly with *Snappy*.
3. In most cases, using Snappy compression improves the performance on both file formats with Hive, except for the OpenNLP query type, where we observe negative influence (Section 4.3). In particular, the *Compression Improvement (CI%)* for Parquet is negative with a value of −18.4%.

5 | SPARK SQL

In the next sections, we show results for the Spark SQL processing engine, again divided in the four BigBench query types as listed in Table 6. The first column reports the query number, whereas the following columns report execution times for each combination of configuration (Table 7) and file format type. The *Total Sum* of query execution times is shown at the end of the table, whereas the very last two rows report the *PI%* and *CI%*.

Query execution time is reported in seconds, and green cells highlight the best time between file formats within each of our configurations. Pairs of cells with the same color (white) are used to show similar performance between the two file formats. **We consider query performance comparable/equal if the difference is lower or equal to** 5% **of the highest execution time.** The standard deviation in % between the 3 query executions for all SparkSQL queries varies greatly. Queries Q02 and Q30 achieve standard deviations varying between 7% and 16%, which will be explained in Section 5.4. All other queries have standard deviations around 10%, which indicates that SparkSQL is less stable than Hive as reported in the works of Ivanov and Beer.[24,25] We believe this is also due to *execution noise* in the cluster affecting more SparkSQL times that are generally much shorter compared to Hive ones.

Similar to the Hive engine, we will first discuss the performance of the *Default* configuration. Next, we observe *No compression* and *Snappy* configurations to get insights and better understand the performance behavior. Generally, query performance on Spark showed more variable results. This is also due to shorter execution times with respect to Hive and therefore bigger noise influence. The query source code for the Spark experiments is exactly the same as the one in Hive, and therefore, we use the same query groupings in this section.

5.1 | Pure HiveQL

Table 13 reports results for the *Pure HiveQL* query type. Figure 8 depicts the query execution times as visual chart. In general, Parquet seems to perform better in all the three configurations.

Looking at the *Default* configuration, Parquet performs better than ORC for many of the queries despite the former does not use compression and the latter uses ZLIB. Some queries perform similarly on both file formats (ie, Q07, Q11, and Q21), whereas other achieve considerably lower execution times with ORC (ie, Q16, Q22, and Q24). In particular, Q22 shows a huge improvement of ~79.3% when run on ORC with the Default configuration compared to the *No compression* and *Snappy* configurations. Further experiments with ZLIB compression need to be performed in order to understand this behavior, but this is out of the scope of our work.

Moving to the *No compression* column, it is more clear that, when the two file formats are under the same conditions, Parquet is the optimal choice. At the same time, a group of six queries show similar performance between the two formats (ie, Q14, Q15, Q21, Q22, Q23, and Q24). The PI% is significant with Parquet taking ~21.5% less time than ORC to execute the full set of queries.

On the *Snappy* column, we can observe that both file formats get a performance benefit from the application of data compression. Parquet gets an improvement of ~5.5% (orange cells), whereas ORC gets a more modest improvement of ~1.5%. Parquet clearly demonstrates its

TABLE 13 Spark Results for *Pure HiveQL* query type. Query execution time is reported in seconds. Green cells, as well as <> symbols, highlight the best time between file formats within each configuration. Pairs of cells with similar performance are filled with the same color (white or light gray) and divided by the ≈ symbol. *PI%* and *CI%* respectively show the performance difference between formats and between uncompressed and compressed configurations

	Default (sec.)			No compression (sec.)			Snappy (sec.)		
Query	ORC		Parquet	ORC		Parquet	ORC		Parquet
Q06	894	>	467	894	>	480	884	>	475
Q07	114	≈	118	130	>	123	116	≈	117
Q09	202	>	136	199	>	140	201	>	135
Q11	126	≈	121	134	>	116	117	>	110
Q12	331	>	105	331	>	118	314	>	122
Q13	278	>	155	274	>	176	271	>	150
Q14	148	>	138	138	≈	140	143	>	126
Q15	141	>	118	133	≈	132	139	>	125
Q16	451	<	613	677	>	630	665	≈	636
Q17	254	>	163	245	>	168	246	>	155
Q21	270	≈	262	244	≈	232	233	>	210
Q22	135	<	654	680	≈	688	687	>	635
Q23	155	>	145	167	≈	160	170	>	132
Q24	164	<	174	166	≈	160	164	>	154
Total Time (sec.)	3662	>	3369	4412	>	3463	4349	>	3282
Performance Improvement (PI) %	⟶		8.0 %	⟶		21.5 %	⟶		24.5 %
Compression Improvement (CI) %	20.5 %		baseline	baseline		baseline	1.5 %		5.5 %

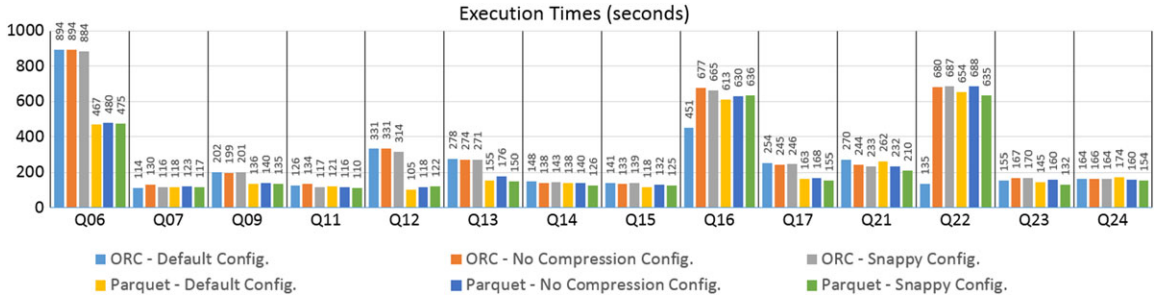

FIGURE 8 Hive Results (in minutes) for the Pure HiveQL queries with all three file format configurations

performance advantages over ORC in all queries with the exceptions of Q07 and Q16 that perform similarly. The PI% between the two formats is even more significant than the *No compression* case, reaching ~24.5%.

5.2 | MapReduce/Python

Table 14 reports results for the *MapReduce/Python* query type. Figure 9 depicts the query execution times as visual chart.

TABLE 14 Spark Results for *MapReduce/Python* query type. Query execution time is reported in seconds. Green cells, as well as <> symbols, highlight the best time between file formats within each configuration. Pairs of cells with similar performance are filled with the same color (white or light gray) and divided by the ≈ symbol. *PI%* and *CI%* respectively show the performance difference between formats and between uncompressed and compressed configurations

	Default (sec.)			No compression (sec.)			Snappy (sec.)		
Query	ORC		Parquet	ORC		Parquet	ORC		Parquet
Q01	137	>	127	127	<	146	122	≈	128
Q02	15644	<	19287	19471	>	15937	19441	>	17580
Q03	2709	≈	2683	2750	≈	2737	2713	≈	2653
Q04	6319	<	7062	7375	≈	7032	6727	≈	6752
Q08	913	>	675	911	>	712	900	>	639
Q29	203	<	504	197	<	517	199	>	184
Q30	6837	<	15253	8690	<	11588	9368	<	11539
Total Time (sec.)	32762	<	45592	39522	>	38669	39471	≈	39474
Performance Improvement (PI) %	28.1 %		⟵	⟶		2.2 %	0 %		⟵
Compression Improvement (CI) %	20.6 %		baseline	baseline		baseline	0.1 %		-2.0 %

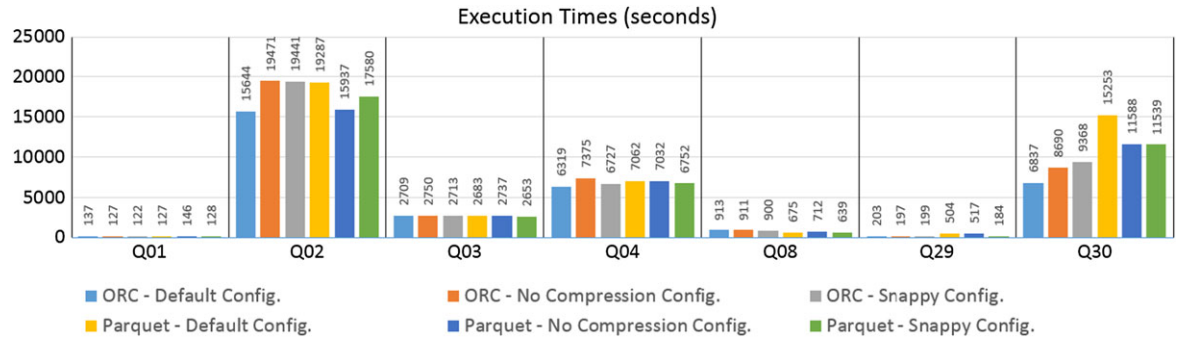

FIGURE 9 Hive Results (in minutes) for the MapReduce/Python queries with all three file format configurations

First, we need to say that Q02 and Q30 are highly unstable on Spark when using the Parquet file format. The two queries randomly fail with *"Out of Memory"* errors, but in many cases, they complete successfully. To obtain the results in Table 14, we ran Q02 and Q30 more than three times with Parquet until we were able to get 3 successful run completions. We report the results for completeness, but we do not consider them trustworthy for our discussion. We did not observe this behavior with any of the ORC configurations.

With the *Default* configuration, ORC performs better with the majority of the queries except Q01 and Q08 performing better on Parquet and Q03 showing similar behavior. The PI% shows a ∼28.1% difference between the two formats. One reason for this behavior is that Q30 on Parquet is taking more than twice the time needed to execute it on ORC. We observed that Spark uses a lot more memory with Parquet and often fails with the aforementioned errors.

Moving to the *No compression* column, there is no clear winner between ORC and Parquet. Q08 performs better on Parquet, whereas Q01 and Q29 perform better on ORC. Q03 and Q04 show similar behavior on both file formats.

For the *Snappy* configuration, we cannot say which file format performs better. It is interesting to observe that the CI% achieved with the introduction of Snappy compression is negligible for ORC (∼0.1%) and even negative for Parquet (∼ −2.0%). We should keep in mind that the total times are highly influenced by the unstable behavior of Q02 and Q30 on Parquet.

Finally, it is hard to conclude anything meaningful for the *MapReduce/Python* query type on Spark. By changing the file format configuration, many queries show no change in their performance (Q03 and Q04), whereas others show contradictory behavior (Q29 and Q01). Only Q08 performs always better on Parquet for all the configurations.

5.3 | HiveQL/OpenNLP

Table 15 reports results for the *HiveQL/OpenNLP* query type. Figure 10 depicts the query execution times as visual chart.

On the *Default* column, execution time is shorter on Parquet because it uses uncompressed data as default parameter. Q19 shows no change in performance and this behavior is confirmed also in the other columns of the table. In general, Q19 performance looks not to be affected by any change in the file format configuration.

Similar to the Hive behavior in Section 4.3, we can observe that this set of queries in Spark SQL performs better on non-compressed data for both file formats. Comparing *No compression* and *Snappy*, we can observe how enabling the data compression decreases the performance for all queries and for both file formats. In fact, the *CI%* is negative for both ORC (blue cells) and Parquet (orange cells).

Looking at the *No compression* column, Q19 and Q27 show comparable performance, whereas Q10 and Q18 perform better on Parquet. The *PI%* of ∼10.7% is in favor of Parquet.

TABLE 15 Spark Results for *HiveQL/OpenNLP* query type. Query execution time is reported in seconds. Green cells, as well as <> symbols, highlight the best time between file formats within each configuration. Pairs of cells with similar performance are filled with the same color (white or light gray) and divided by the ≈ symbol. *PI%* and *CI%* respectively show the performance difference between formats and between uncompressed and compressed configurations

	Default (sec.)			No compression (sec.)			Snappy (sec.)		
Query	**ORC**		**Parquet**	**ORC**		**Parquet**	**ORC**		**Parquet**
Q10	1994	>	976	2026	>	1633	2070	<	3075
Q18	2711	>	2011	2699	>	2519	2836	≈	2721
Q19	521	≈	525	526	≈	530	535	≈	537
Q27	140	>	103	138	≈	128	139	≈	145
Total Time (sec.)	5366	>	3615	5388	>	4811	5581	<	6477
Performance Improvement (PI) %	→		32.6 %	→		10.7 %	13.8 %		⟵
Compression Improvement (CI) %	0.4 %		baseline	baseline		baseline	-3.5 %		-25.7 %

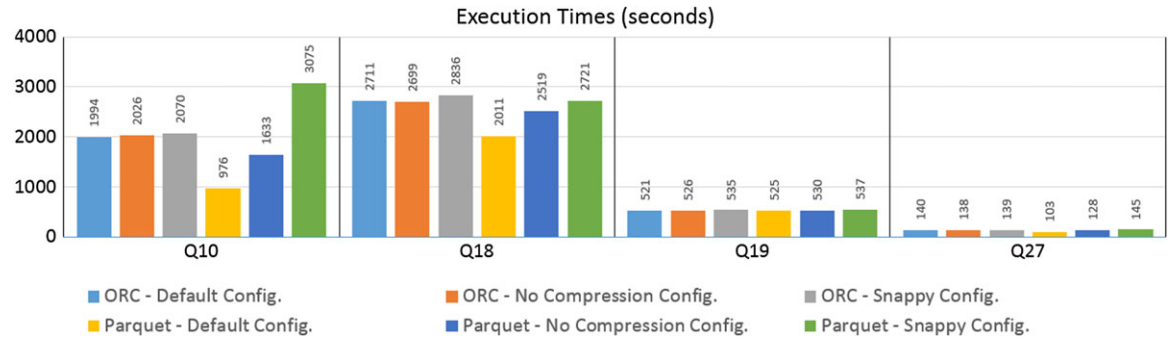

FIGURE 10 Hive Results (in minutes) for the HiveQL/OpenNLP queries with all three file format configurations

As stated above, both file formats performance is decreasing with the *Snappy* configuration compared to the *Default* and *No Compression*. It is interesting to observe that Q10 is faster with ORC and *Snappy* compared to Parquet and *Snappy*.

5.4 | HiveQL/Spark MLlib

Table 16 reports results for the *HiveQL/Spark MLlib* query type. Figure 11 depicts the query execution times as visual chart.

Looking at the *Default* configuration, ORC achieves a better *Total Time*, with a *PI%* of ~36.9%. Q20 and Q25 perform better on Parquet, whereas Q28 does not show performance differences for any of the configurations.

Moving to the *No compression* column, ORC is still the best option in terms of total time but the *PI%* between the two formats is similar. The most unusual behavior has Q05. It takes 618 seconds on ORC *Default* (using ZLIB compression) compared to 1842 seconds on ORC *No compression*. All other queries do not seem to be affected by the configuration changes.

With *Snappy* data compression, the two formats show similar performance in all queries, with the exception of Q20 and Q25 that perform better on Parquet. While Parquet gets a significant improvement thanks to data compression (~7.0%, orange cells), ORC gets a minimal improvement in the *Total Time*. Only queries Q05 and Q25 perform better when using ORC with Snappy instead of *No Compression*, whereas all the remaining queries (Q20, Q26, Q28) result in longer execution times.

TABLE 16 Spark Results for *HiveQL/Spark MLlib* query type. Query execution time is reported in seconds. Green cells, as well as <> symbols, highlight the best time between file formats within each configuration. Pairs of cells with similar performance are filled with the same color (white or light gray) and divided by the ≈ symbol. *PI%* and *CI%* respectively show the performance difference between formats and between uncompressed and compressed configurations

	Default (sec.)			No compression (sec.)			Snappy (sec.)		
Query	**ORC**		**Parquet**	**ORC**		**Parquet**	**ORC**		**Parquet**
Q05	618	<	1825	1842	≈	1900	1811	≈	1787
Q20	343	>	298	343	>	299	345	>	294
Q25	462	>	359	453	>	363	440	>	346
Q26	272	<	374	266	<	365	281	≈	290
Q28	304	≈	309	297	<	326	308	≈	322
Total Time (sec.)	1998	<	3165	3201	≈	3252	3187	≈	3039
Performance Improvement (PI) %	36.9 %		⟵	1.6 %		⟵	→		4.6 %
Compression Improvement (CI) %	60.2 %		baseline	baseline		baseline	0.5 %		7.0 %

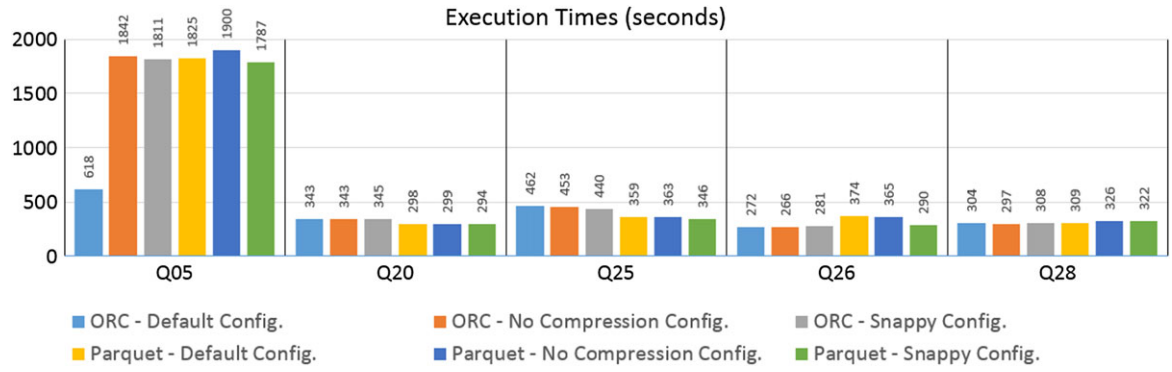

FIGURE 11 Hive Results (in minutes) for the HiveQL/SparkMLlib queries with all three file format configurations

5.5 | Summary

Next, we summarize the major findings of all experiments reported in this section.

1. Using the "out-of-the-box" *Default* file format configuration is not always the optimal choice and depends heavily on the data structure and query type (Section 5.3).
2. Parquet generally achieves best performance with the Spark engine except for *HiveQL/Spark MLlib* (Section 5.4) and *MapReduce/Python* (Section 5.2) query types, where behavior is unclear.
3. In most cases, using Snappy compression improves the performance on both file format with Spark, except for the OpenNLP query type, where we observe negative influence (Section 5.3). In particular, the *Compression Improvement (CI%)* for Parquet is negative with a value of −25.7%.
4. Queries Q02 and Q30 perform unstable on Spark (Section 5.2).

6 | IN-DEPTH QUERY ANALYSIS

In this section, we perform a deeper query investigation with the goal to identify the causes of the varying performance behavior reported in the previous sections. Due to time limitations, we select one representative query from each of the four query types in BigBench. These are Q08 (MapReduce/Python), Q10 (HiveQL/OpenNLP), Q12 (Pure HiveQL), and Q25 (HiveQL/Spark MLlib).

We re-executed the queries on SparkSQL configured with *No compression* and *Snappy* for ORC and Parquet. Each of them shows different behavior with respect to the other queries in the same group. We analyze them with the help of the Spark History Server metrics, the query source code, and the resource utilization data. Figure 12 shows all the information we collected about a query (ie, Q08 with *No Compression* on ORC) by looking at the Spark History Server and source code. In particular, we collect the Spark tasks, stages, operations, as well as Input and Output data sizes. A Spark query can be split into a number of jobs. A Spark job is a set of tasks that is resulting from Spark operation (action). A task is an individual unit of work that is executed on one executor/container. A Spark stage is a set of tasks in a job that can be executed in parallel. The operations are either actions or transformations executed in a job (ie, HadoopRDD or FileScanRDD). The Input size is the data read in a stage, whereas the Output size is the data written/resulting from a stage. More details about the Spark internals are available in the Spark documentation.[66] Using this information, we compiled summary tables for each query execution to better understand the unexpected behavior and compare the different file format configurations. The complete tables are available on GitHub[65] under file *Evaluation-summary.xlsx*. In addition, we collected performance metrics using the Performance Analysis Tool (PAT)[67] by Intel. The tool collects data from all cluster nodes introducing minimal overhead on the benchmark workload. At the end, it aggregates them and immediately produces Excel files with visual charts, which will be used in our analysis. All generated charts are available on GitHub[65] under file *selected-queries-resource.pdf*. In the next subsections, we discuss our findings for each of the selected queries.

6.1 | BigBench Q08 (*MapReduce/Python*)

Q08 investigates the effect of review reading by customers on the sales revenue in a specific time and for a specific product category. The query can be divided in two distinct phases. In the first phase, three temporary tables are created and stored in plain text format. The first table contains rows representing dates for the desired time period. The second stores web sessions of users who read product reviews. A Python program is used to build this. The third and last table stores sales in the desired time period. In the second phase, the temporary tables are combined to get the final result. To better understand the file format impact, we focus primary on the first phase where the data is retrieved from the file format structure.

6.1.1 | Comparing ORC and Parquet

As shown in Table 14, Q08 performs better on Parquet for both *No Compression* and *Snappy* configurations. Similarly for both configurations, Parquet reads less input data for the key execution stages compared to ORC.

No Compression Configuration: The Parquet execution (12 min.) is faster than the ORC execution (16 min.). The ORC execution results in 14 Spark stages with total *Input* of 111.6 GB compared to 18 Spark stages with total *Input* of 94.5 GB for the Parquet execution. In ORC, stage 3 (5.8 min.) performs the same operations as stage 6 (2.8 min.) in Parquet. Both stages execute 1402 tasks. However, ORC stage 3 takes as *Input* 95.7 GB and *Outputs* 15.2 GB, whereas Parquet stage 6 takes 77.8 GB for the same *Output*.

Figure 13 shows the CPU utilization (blue peaks in the yellow square) for the *No Compression* configuration. The first block of lines showing high CPU utilization should correspond to the first phase. Here, relevant data for the query is retrieved by using file format indexes and metadata. ORC spends more CPU time for User processes with respect to Parquet. For the latter, the majority of CPU time in the first phase is spent waiting for I/O (disk) operations. This can be interpreted as Spark being better in retrieving relevant data from Parquet and immediately triggering I/O requests.

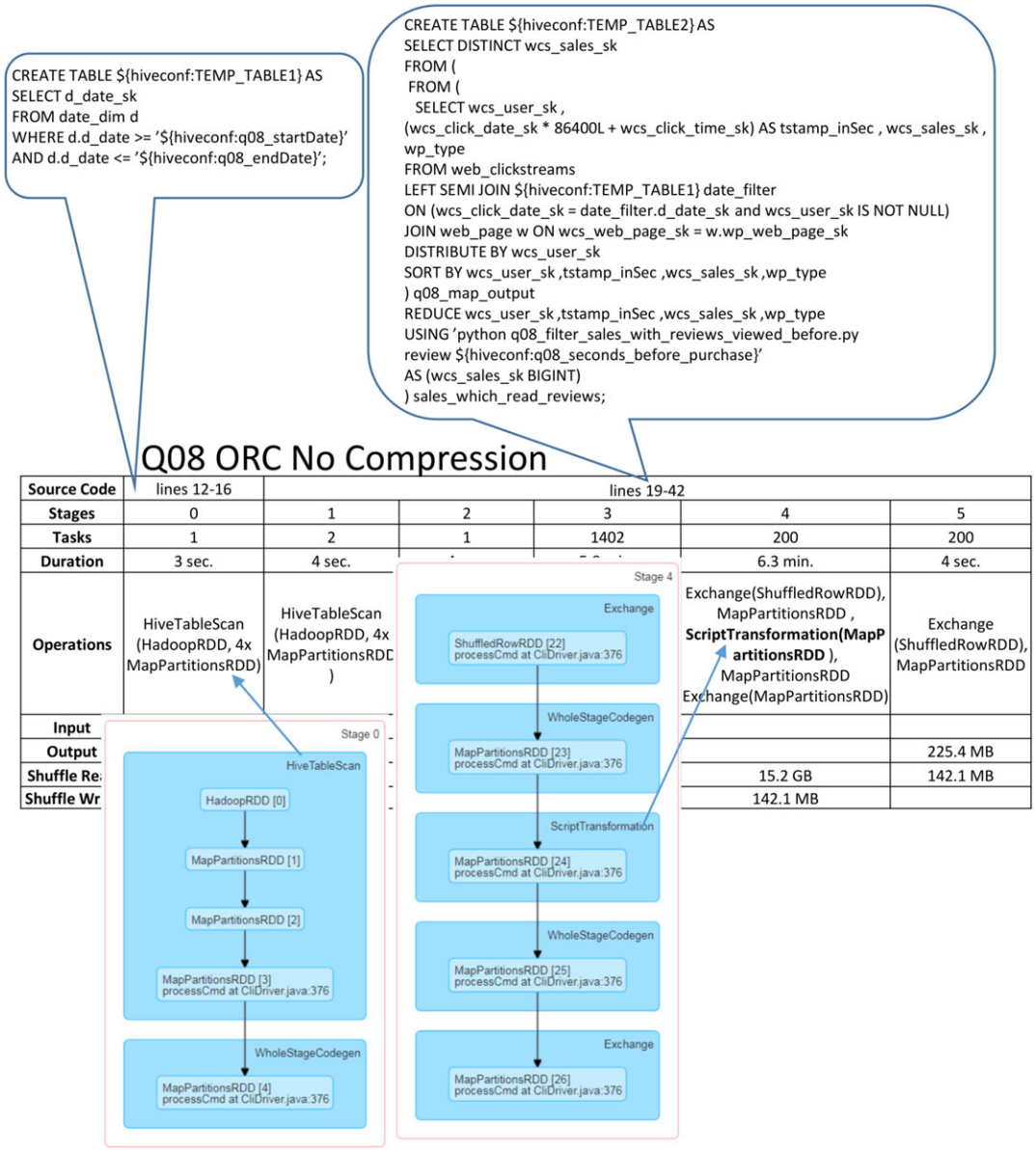

FIGURE 12 Illustrates the process of collecting the Spark History Server metrics and source code details into a summary table for query Q08 executed on ORC configured with No Compression

Snappy Configuration: The Parquet execution (11 min.) is faster than the ORC execution (15 min.). Both formats execute in 14 Spark stages. Compared the total *Input* for ORC is 73.7 GB, whereas, for Parquet, it is 61.8 GB. In ORC, stage 3 (5.5 min.) performs the same operations as stage 3 (2 min.) in Parquet. The number of executed tasks is almost equal with 1402 for ORC and 1401 for Parquet. However, the ORC stage 3 takes 61.6 GB as *Input*, whereas the Parquet takes only 48 GB, with *Output* for both 15.2 GB. With respect to executed operations, ORC performs one *HadoopRDD* and 5 *MapPartitionsRDD* operations, whereas Parquet performs one *FlieScanRDD* and 2 *MapPartitionsRDD* operations.

6.1.2 | ORC

Execution with *No Compression* configuration takes 16 min., whereas with Snappy configuration takes 15 min. Both configurations execute for 14 Spark stages. Compared the total *Input* for *No Compression* configuration is 111.6 GB, whereas the total *Input* for *Snappy* configuration is 73.3 GB.

6.1.3 | Parquet

Execution with *No Compression* configuration takes 12 min., whereas with *Snappy* configuration takes 11 min.. *No Compression* configuration is executed for 18 stages, whereas the *Snappy* configuration for 14 stages. Compared the total *Input* for *No Compression* configuration is 94.5 GB, whereas the total *Input* for *Snappy* is 61.8 GB.

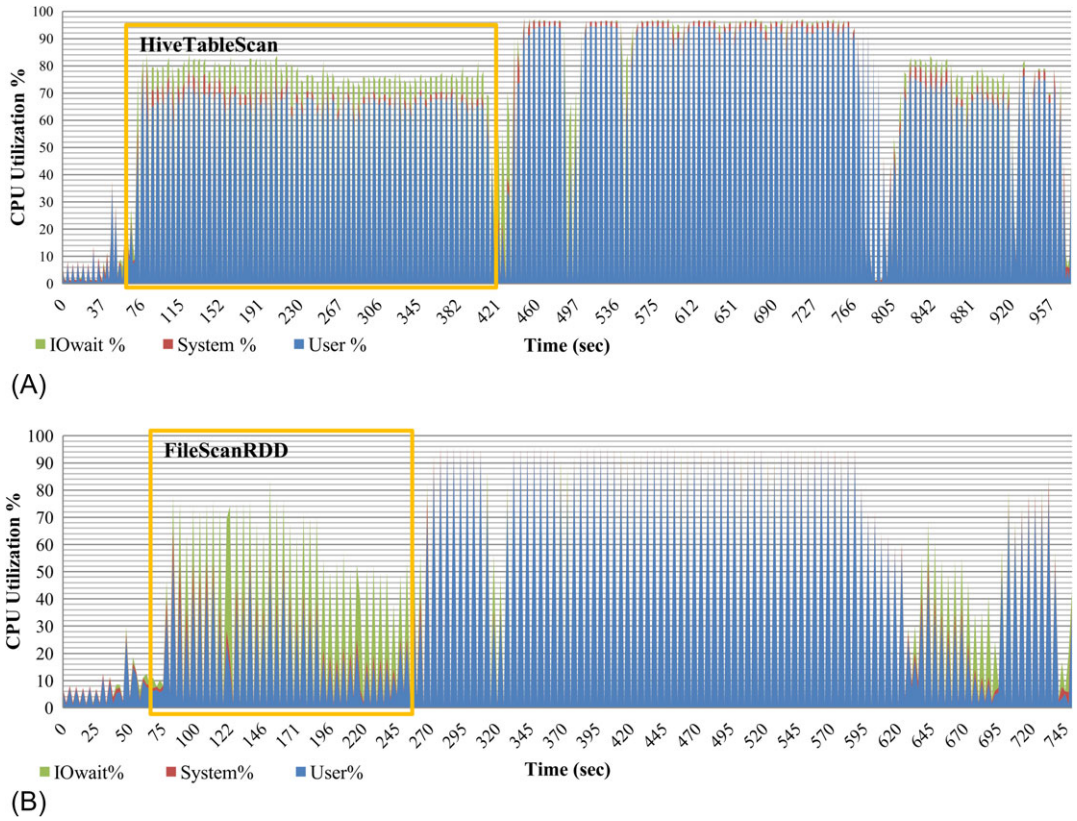

FIGURE 13 CPU utilization for Q08 with *No Compression* configuration. A, ORC; B, Parquet

The use of *Snappy* compression brings some improvements only for Parquet. Figure 14 shows a huge drop in I/O requests (blue peaks in the yellow square). This is the expected behavior when using compression: to reduce the number of disk accesses while retrieving the same data.

6.2 | BigBench Q10 (*HiveQL/OpenNLP*)

Q10 performs sentiment analysis on the product reviews by classifying them as *positive* or *negative*. The output also reports the words that lead to the classification. The query does not create any temporary table, but it works directly on data stored in columnar file formats. The sentiment extraction is realized with a Java UDF embedded in the query code.

6.2.1 | Comparing ORC and Parquet

This query type is particularly interesting because it generally performs better with *No compression* configuration, as shown in Table 15. This is an unexpected behavior, as the data compression usually brings benefit to the performance by reducing the I/O accesses. In the specific case of Q10, we get better performance on Parquet when using uncompressed format. On the other hand, the query performs better with *Snappy* compression on ORC, but the performance improvement is less relevant.

No Compression Configuration: The Parquet execution (29 min.) is faster than the ORC execution (34 min.). The ORC execution results in 3 Spark stages, whereas the Parquet execution results in 4 Spark stages. In terms of total *Input* data, both take 5 GBs. However, in ORC, stage 0 executes on 9 tasks for 16 min., compared to Parquet stage 1 that executed on 28 tasks for 14 min.. Similarly, in ORC, stage 1 executed on 9 tasks for 18 min., compared to Parquet stage 2 that executed on 28 tasks for 14 min.. The ORC stage 0 performs one *HadoopRDD* and 7 *MapPartitionsRDD* operations, whereas Parquet stage 1 performs one *FileScanRDD* and 5 *MapPartitionsRDD* operations. The ORC stage 1 performs one *HadoopRDD* and 5 *MapPartitionsRDD* operations, whereas Parquet stage 2 performs *FileScanRDD* and 3 *MapPartitionsRDD* operations. We can conclude that Parquet takes advantage of parallel execution by performing 28 tasks in parallel compared to 9 tasks for ORC.

Snappy Configuration: The ORC execution (37 min.) is faster than the Parquet execution (51 min.). The ORC executions results in 3 Spark stages, whereas the Parquet execution results in 4 Spark stages. ORC takes as total *Input* 3.4 GB, whereas Parquet takes as total *Input* 4.7 GB.

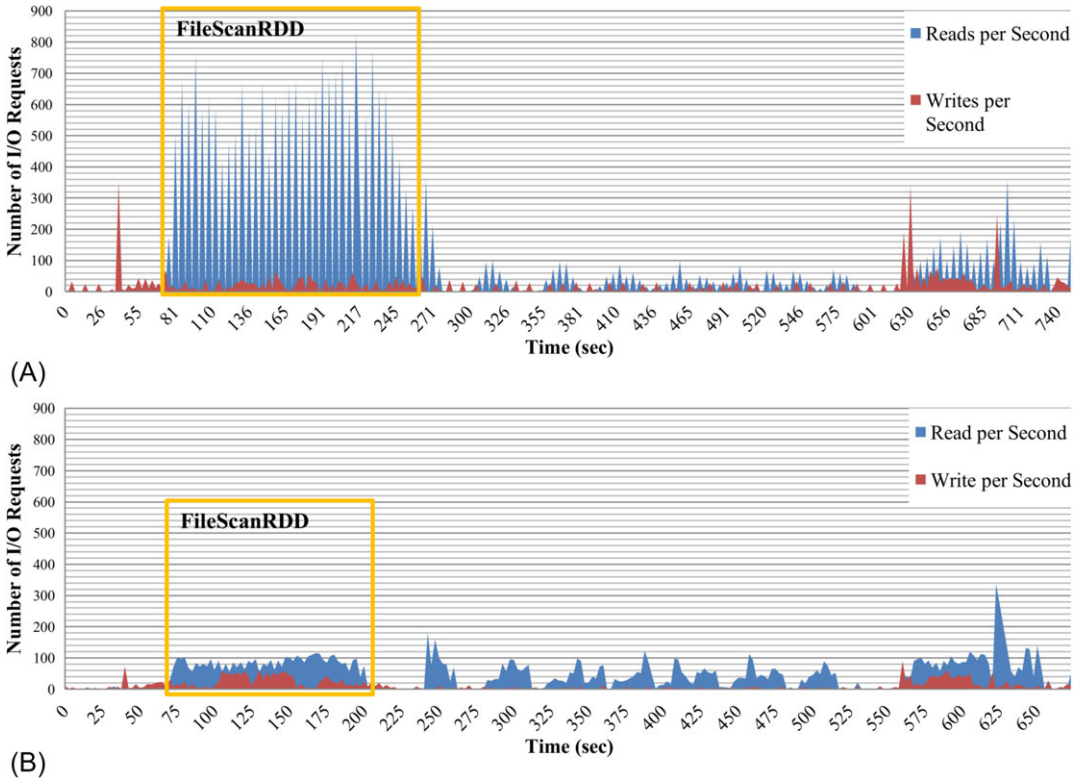

FIGURE 14 Disk Requests for Q08 with Parquet. A, No Compression, B; Snappy

The biggest difference is that stage 1 in Parquet is not comparable to any of the ORC stages. Parquet stage 1 takes 15 min. and performs one *FileScanRDD*, 4 *MapPartitionsRDD*, one *PartitionPrunningRDD*, and one *PartitionwiseSampleRDD*, which are not performed in ORC. Both operations *PartitionPrunningRDD* and *PartitionwiseSampleRDD* are not typical for this Spark context and lead to unnecessary increase of the total execution time. Therefore, further investigation for the cause of this behavior needs to be done.

For what concerns Parquet, we hypothesize that Q10 is performing better on *No compression* configuration because it is harder to efficiently compress unstructured data. Anyway, the behavior is not confirmed when using ORC. Because of this discrepancy, we look into the resource utilization. Figure 15 shows the disk requests for the *No compression* configuration. We can immediately notice that the disk is underutilized with both file formats. The same happens with *Snappy* configurations and also for the disk bandwidth utilization and CPU utilization, which we do not report in this document for the sake of space.

In this case, the motivation for the query performance behavior is unclear. If the benchmark user is specifically interested in this query type, he should further investigate Q18, Q19, Q27. If no relevant insight is found, the UDF code should be checked for implementation issues.

6.2.2 | ORC

The *Snappy* configuration takes 37 min. and total input of 5 GB compared to 34 min. and total input of 3.4 GB with *No Compression* configuration. Beside the smaller input data, using compression results in worse performance than with *No Compression*.

6.2.3 | Parquet

The *Snappy* configuration takes 51 min. and 4.7 GB total *Input* compared to 29 min. and 5 GB total input with the *No Compression* configuration. Both configurations have 4 Spark stages, but stage 1 with *Snappy* is not performing the same operations as already mentioned above. Beside the smaller input data, using compression results in worse performance than with *No Compression*.

6.3 | BigBench Q12 (*Pure HiveQL*)

Q12 searches for customers who viewed products online and then bought a product from the same category in a physical store in the next three months. No temporary tables are created to fulfill the query, meaning that all the relevant data is retrieved directly from the columnar file format. The whole query period should be relevant for the analysis.

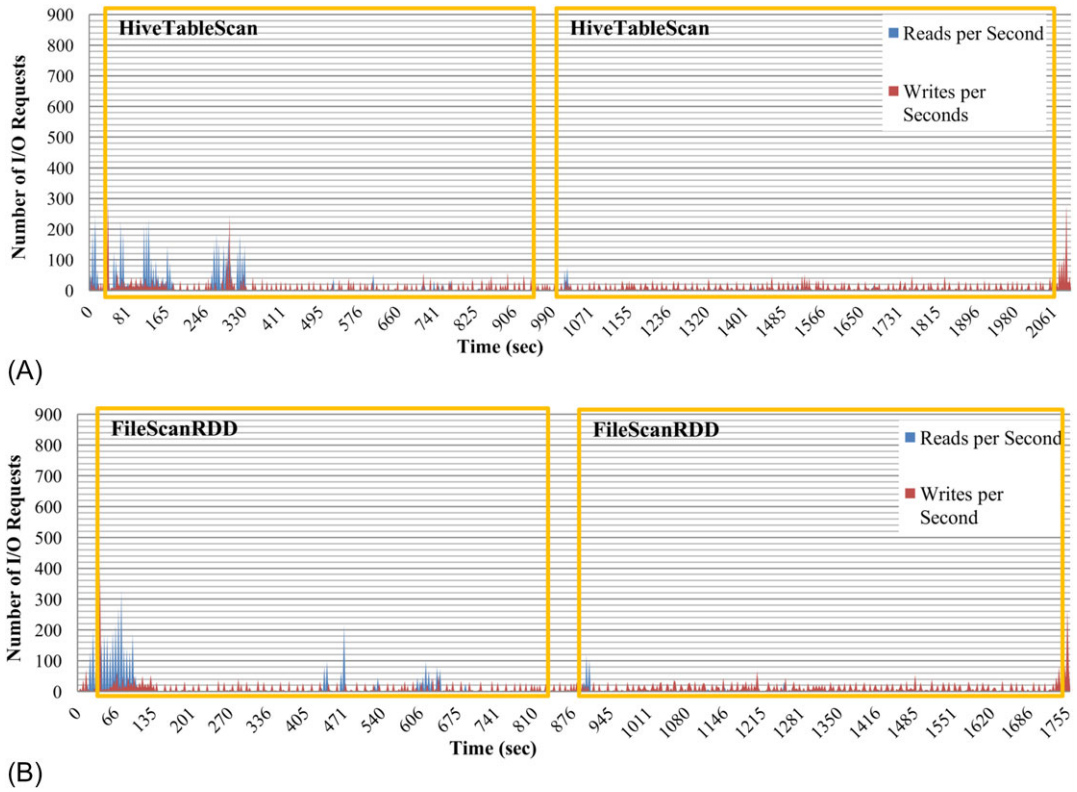

FIGURE 15 Disk requests for Q10 with *No Compression* configuration. A, ORC; B, Parquet

6.3.1 | Comparing ORC with Parquet

Table 13 shows that Q12 always performs better on Parquet compared to ORC. *Snappy* compression brings a slight improvement for ORC, while it worsens a bit the Parquet performance. It is the only query that does not get a performance improvement on the *Snappy* configuration with Parquet. For both configurations, Parquet reads much less input data compared to ORC, which has a clear impact on the performance.

No Compression Configuration: The Parquet execution (1.7 min.) is faster than the ORC execution (5.4 min.). The ORC execution is in 8 Spark stages with total *Input* of 82.6 GB, whereas the Parquet execution is in 10 stages with 2.9 GB total *Input*. Both are executed on 1402 Spark tasks. The ORC stage 1 takes 3.8 min. compared to the Parquet stage 4 taking 15 sec..

Snappy Configuration: The Parquet execution (1.4 min.) is faster than the ORC execution (5.2 min.). Both formats execute in 8 Spark stages. The ORC stage 1 takes 3.6 min. with 40.9 GB total *Input* data compared to the Parquet stage 1 taking 11 sec. with 1172 MB total *Input*. Similarly, the ORC stage 2 takes 32 sec. and 4.4 GB total *Input* compared to the Parquet stage 2 taking 4 sec. and 354 MB total *Input*.

6.3.2 | Optimized record columnar

Using *Snappy* compression takes 5.2 min. and 45.3 GB compared to *No Compression* with 5.4 min. and 82.6 GB. Overall, using compression with ORC slightly improves the performance.

6.3.3 | Parquet

Using *Snappy* compression, the query takes 1.4 min and 1.5 GB compared to *No Compression* configuration with 1.7 min. and 2.9 GB. Overall, using compression with Parquet slightly improves the performance. As the reader can notice, measurements in the run with PAT show a different result with respect to Table 13.

To understand the aforementioned discrepancy, we look at the PAT utilization charts. The disk requests and bandwidth utilization profiles are very similar between *No compression* and *Snappy* configurations. We do not report them here since we cannot get any useful insight from them. Figure 16 shows the CPU utilization for Q12 on the Parquet file format for both configurations. We observe a generally higher utilization with

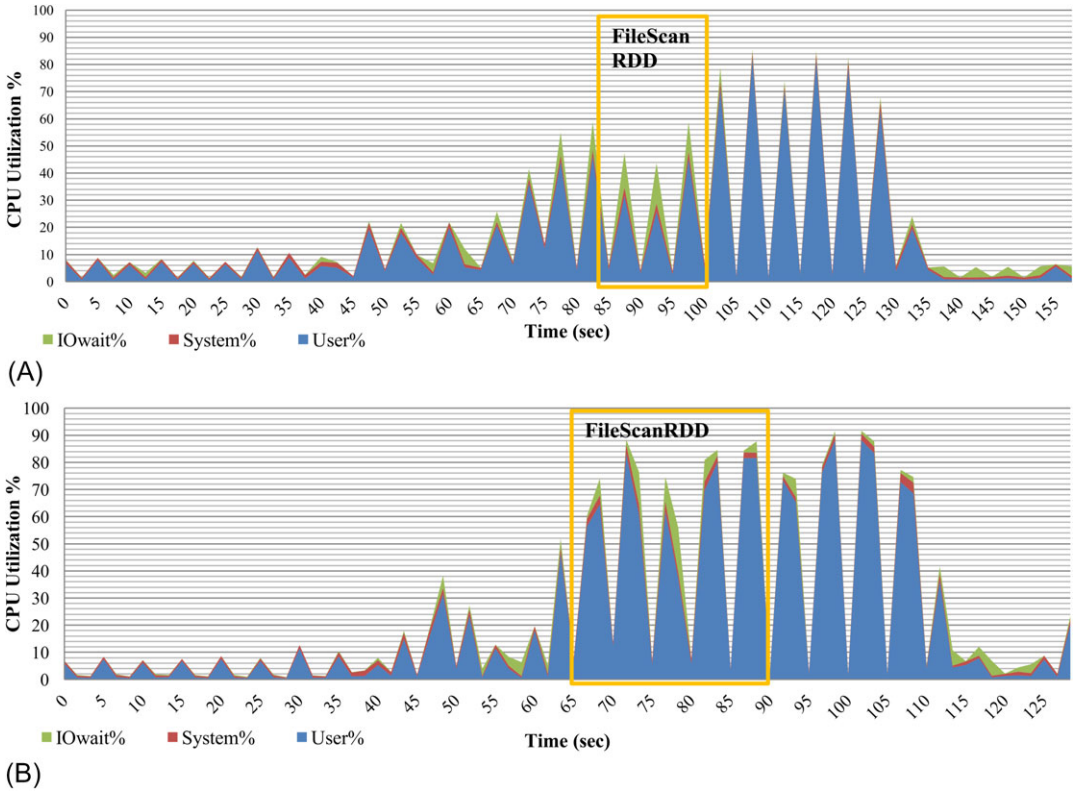

FIGURE 16 CPU utilization for Q12 with Parquet. A, No compression; B, Snappy

Snappy configuration. This is usually expected since data retrieved from disk must be uncompressed, wasting valuable CPU time. As observed in all the other queries in Table 13, despite the need of extra computation, data compression is beneficial for the performance since it reduces the number of disk accesses. The unexpected behavior of Q12 can be explained with the fact that the query reads small amount of data from the disk; the benefit from using compression is not appreciable. We believe that the discrepancy between Table 13 and PAT execution is due to noise, and the difference between *No Compression* and *Snappy* configuration should be considered negligible.

6.4 | BigBench Q25 (*HiveQL/Spark MLlib*)

Q25 groups customers based on a set of shopping dimensions. To achieve this, it uses a k-means clustering algorithm[68] implemented using the Spark MLlib. The query is split into two phases: the first phase creates a temporary table to prepare data for the clustering algorithm, whereas the second phase simply executes a Spark job and stores results. As usual, the temporary table is stored in plain text making only the first phase relevant for our performance analysis on file formats.

6.4.1 | Comparing ORC and Parquet

Table 16 reports that Q25 always performs better on Parquet for both *No compression* and *Snappy* configurations. The data compression brings little improvement when using Parquet (~4.7%), whereas ORC gets a minimal improvement. Other queries in Table 16 show worse performance on ORC for *Snappy* configuration with respect to *No compression*.

No Compression Configuration: The Parquet execution (3.4 min.) is faster than the ORC execution (5 min.). Both execute in 13 Spark stages and the same number of tasks. The ORC execution takes as *Input* 18.3 GB, whereas the Parquet execution takes as total *Input* 23.8 GB. However, ORC stages 1 and 5 execute *HadoopRDD* and 5 *MapPartitionsRDD*, whereas Parquet stages 1 and 5 execute *FileScanRDD* and 2 *MapPartitionsRDD*.

Snappy Configuration: The Parquet execution (3.3 min.) is faster than the ORC execution (5 min.). Both execute in 13 Spark stages and the same number of tasks. The ORC execution takes 10.7 GB as total *Input*, whereas the Parquet execution takes 15.7 GB as total *Input*. Again, ORC stages 1 and 5 execute *HadoopRDD* and 5 *MapPartitionsRDD*, whereas Parquet stages 1 and 5 execute *FileScanRDD* and 2 *MapPartitionsRDD*.

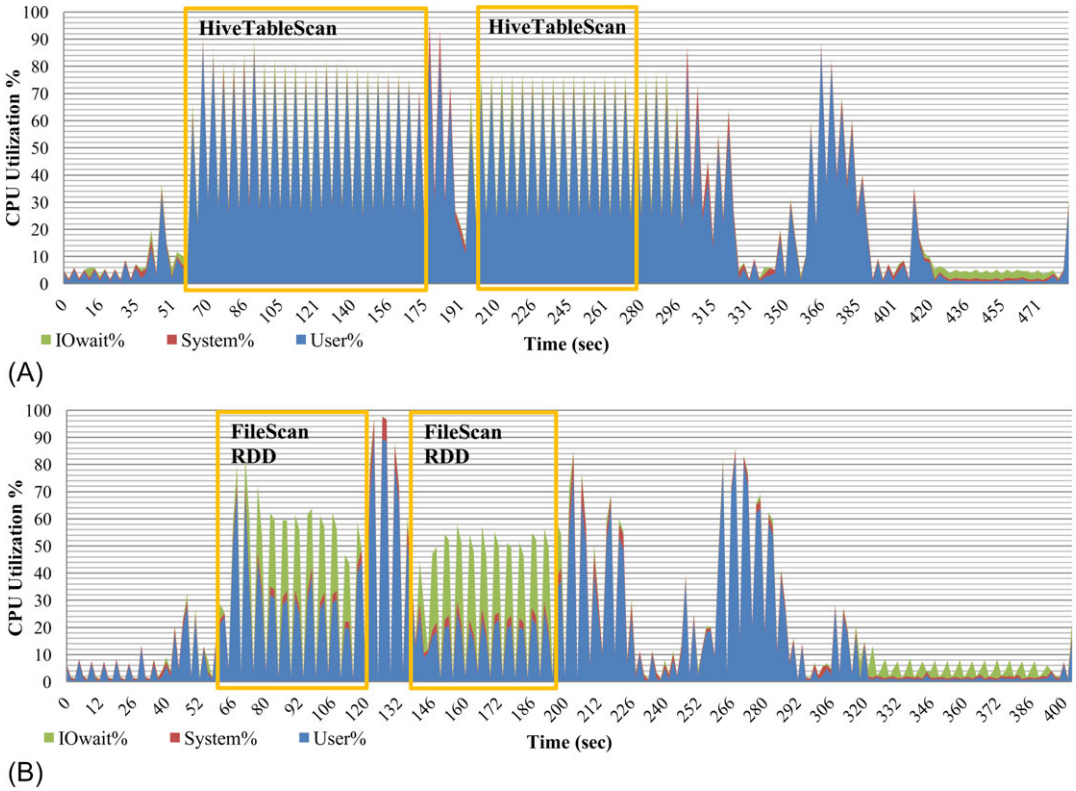

FIGURE 17 CPU utilization for Q25 with *Snappy* configuration. A, ORC; B, Parquet

Figure 17 compares the CPU utilization on ORC and Parquet for *Snappy* configuration. As already observed for Q08, more CPU user time (blue peaks in the yellow square) is spent on ORC in the data retrieving phase. It looks like that, on Parquet, it is easier for Spark to address relevant data and request it from the disk. In fact, CPU user time is lower on Parquet, whereas a large part of the total CPU time is spent for I/O wait (indicated by the green color in the yellow squares).

6.4.2 | ORC

ORC performs equal with both *Snappy* and *No Compression* configurations taking 5 min., 13 stages, and equal number of tasks with 18.3 GB total *Input* for *No Compression* and 10.3 GB total *Input* for *Snappy*. Overall, using compression with ORC do not improve the performance.

6.4.3 | Parquet

Similar to ORC, Parquet also performs equal with both *Snappy* and *No Compression* configurations taking around 3.4 min., 13 stages, and equal number of tasks with 23.8 GB total *Input* for *No Compression* and 15.7GB total *Input* for Snappy.

6.5 | Summary

1. The most important lesson learned is that changing the file format influences the overall engine behavior. The functions used to retrieve data are different: with ORC, Spark prefers to use *HiveTableScan*, whereas, with Parquet, it prefers to use *FileScanRDD*. This emphasizes the importance of the file format choice when using a specific engine.
2. In many cases, we observed that both formats vary in the number of Spark stages, tasks, and operations, which they execute for the same HiveQL query code.
3. Generally, the introduction of *Snappy* compression improves the query performance for both file formats by reading less data from disk. However, the query performance is not only influenced by the amount of input data from disk. For Q25, Parquet shows better performance than ORC while still reading more data.

4. The *HiveQL/OpenNLP* Q10 shows abnormal behavior. Cluster resources are under utilized and *Snappy* compression worsens the performance. While the latter can be caused by the unstructured data type, the cluster under utilization can be caused by flawed workload code in the benchmark. Further investigation in the Java UDF source code is necessary.

7 | DISCUSSIONS AND FUTURE WORK

In this paper we evaluated the ORC and Parquet file formats on top of Hive and Spark by varying the format configurations and comparing their performance. However, both Parquet and ORC are developed as independent, general-purpose data file formats and as such are easier to integrate into other engines. In Table 1, at the beginning of this paper, we listed multiple engines which already support both formats. This also means that each of the engines offers a different level of integration for both formats as well as different default parameter configurations. For example, Spark SQL uses Parquet with *Snappy* compression[69] as the default storage format, contrary to Hive which uses Parquet with no compression as a default configuration.

In addition to that, every engine focuses on implementing optimizations using its primary storage file format leaving all other supported formats behind. Therefore, when benchmarking one should explicitly differentiate between ''benchmarking SQL-on-Hadoop engine performance'' and ''benchmarking data file formats performance''. Comparing the performance of different engines does not immediately mean that one is also comparing the storage file format used by the engine.

What are the best practices, methodologies and metrics when comparing different file formats?

Based on our experimental results and benchmarking methodologies that we implemented to compare the file formats performance in this study (in particular ORC and Parquet), we assembled a list of steps that can be used as a best practices guide when testing file formats on a new SQL-on-Hadoop engine:

- Make sure the new engine offers support for the file formats under test (ORC and Parquet).
- Choose a benchmark or test workload representing your use case and suitable for the comparison.
- Set the file format configurations accordingly (with or without compression).
- Generate the data and make sure that it is consistent with the file configuration using the file format (ORC and Parquet) tools.
- Perform the experiments (at least 3 times) and calculate the average execution times.
- Compare the time differences from the two file formats using the *PI %* and *CI %* metrics and make conclusions.
- Select queries compatible with your specific use-case. Execute them while collecting resource utilization data and perform in-depth query evaluation to spot bottlenecks and problems.

One important evaluation factor mentioned above is the benchmark used for stress testing the engines. As reviewed by the related work (Section 2.6), TPC-H and TPC-DS are the most common choices for benchmarking both engine and file format performance. In this paper, we utilize the BigBench benchmark as it is currently the only standard benchmark including structured, unstructured and semi-structured data type as well as machine learning and text processing workloads.

Is BigBench suitable for file format comparisons?

Based on our experimental results (Sections 4, 5 and 6), we can conclude that BigBench is a good choice for comparing file formats on SQL-on-Hadoop engines mainly for two reasons: (1) structured and unstructured data influence the query performance in particularly in combination with compression (Q10 in Section 6.2); and (2) the BigBench variety of 30 different workloads (use cases) divided in four categories based on implementation type.

However, many other questions around benchmarking file formats still remain open:

- Is there a need for a specialized micro-benchmark to better investigate the file format features?
- If yes, what should this benchmark include in terms of data types and operations?
- What are the file format features that such a benchmark should stress (for example block size, compression etc.)?

As a future work, we plan to investigate these questions. For example, further insights can be obtained by running benchmarks on the same processing engine and file format while changing the file format parameters. This would help to better distinguish the influence of each architecture component on the query performance. New metrics can be added to the in-depth query analysis, like the amount of network traffic exchanged by each node in the cluster on relevant TCP and UDP ports, as shown in,[70] to spot bottlenecks and unbalanced workloads.

Resource utilization data collection and the comparison of graphs and execution plans can be standardized and integrated into BigBench for better usability. Data analysis and visualization can then be included in a graphical user interface, like Apache Hue[71] which is highly customizable.

We also plan to evaluate columnar file formats with other benchmarks and dataset types, leading to their adoption in other applications. For example, ORC and Parquet together with Hive or SparkSQL can be used to store and query time-series data, like data coming from sensors.

Software architectures for sensor data analysis can select these technologies for their batch layer to achieve better access performance with respect to simple plain text files on HDFS, or to replace complex NoSQL systems.[72,73] However, dedicated benchmarks are needed to ensure the suitability of columnar file formats for range-based queries, that are commonly used to access time-series data.

8 | CONCLUSIONS

To the best of our knowledge, this is the first study that evaluates the ORC and Parquet file formats on Hive and SparkSQL using the BigBench benchmark as a popular representative of Big Data workloads. From our benchmark results, we can observe that it is important to separate the file format evaluation from the engine. Both components have great influence on the workload and a comparison of their combination is meaningless because we cannot tell what component is really causing a change in performance. In Section 6, we showed how a different file format selection changes the Spark execution behavior. In particular, different functions are used to retrieve data respectively from ORC or Parquet files. We believe that our benchmark methodology in which we keep the processing engine fixed while changing the file format is correct.

At the same time, overall performance on the same engine is greatly influenced by the file format parameters. The default configurations of ORC and Parquet are extremely different and their direct performance comparison can lead to misleading results. Therefore, the file format selection cannot be naively based on the engine preference stated by documentation or by quick-and-dirty benchmarks. A careful understanding of file format parameters, especially the use of data compression, is necessary to make the optimal choice.

Once this is setup, a careful understanding of the benchmark workload is necessary. We showed that ORC generally achieves best performance with the Hive engine (Section 4), whereas Parquet generally achieves best performance with the Spark engine (Section 5). However, this is not true for all BigBench query types with significant exceptions, as shown in Sections 4.3, 5.4, and 5.2. Users should identify and select groups of queries in BigBench that best emulate their use case and then evaluate performance. This will not only reduce time needed to execute benchmarks but will also give more valuable results. In this respect, BigBench is extremely flexible: in this work, we ran the whole set of queries but the user can select and run only a subset of them.

Similarly, the use of data compression is not advised for all query types. In most cases, using Snappy compression improves the performance on both file formats and both engines, except for the OpenNLP query type, where we observe negative influence with both engines (Sections 4.3 and 5.3).

In the future, we plan to perform more experiments with different configurations by modifying other file format parameters and investigate their influence. Furthermore, we plan to understand the cause for instability of queries Q02 and Q30 on Spark (Section 5.2) as well as the resource under utilization found with Q10 (Section 6.2).

ACKNOWLEDGMENTS

This research was supported by the Frankfurt Big Data Lab (Chair for Databases and Information Systems - DBIS) at the Goethe University Frankfurt. Special thanks for the help and valuable feedback to Sead Izberovic, Thomas Stokowy, Karsten Tolle, Roberto V. Zicari (Frankfurt Big Data Lab), Pinar Tözün (IBM Almaden Research Center, San Jose, CA, USA), and Nicolas Poggi (Barcelona Super Computing Center).

ORCID

Todor Ivanov https://orcid.org/0000-0002-9679-8947
Matteo Pergolesi https://orcid.org/0000-0001-9626-1608

REFERENCES

1. Thusoo A, Sarma JS, Jain N, et al. Hive: a warehousing solution over a map-reduce framework. *PVLDB Endow*. 2009;2(2):1626-1629. http://www.vldb.org/pvldb/2/vldb09-938.pdf
2. Thusoo A, Sarma JS, Jain N, et al. Hive-a petabyte scale data warehouse using hadoop. Paper presented at: 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010); 2010; Long Beach, CA.
3. Huai Y, Chauhan A, Gates A, et al. Major technical advancements in apache hive. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data; 2014; Snowbird, UT.
4. Dremel Made Simple with Parquet. 2018. https://blog.twitter.com/2013/dremel-made-simple-with-parquet
5. Apache Parquet. 2018. http://parquet.apache.org/
6. Melnik S, Gubarev A, Long JJ, et al. Dremel: interactive analysis of web-scale datasets. *Proc VLDB Endow*. 2010;3(1):330-339.
7. Armbrust M, Xin RS, Lian C, et al. Spark SQL: relational data processing in spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data; 2015; Melbourne, Australia.

8. Kornacker M, Behm A, Bittorf V, et al. Impala: A modern, open-source SQL engine for hadoop. Paper presented at: 7th Biennial Conference on Innovative Data Systems Research; 2015; Pacific Grove, CA.
9. Olston C, Reed B, Srivastava U, Kumar R, Tomkins A. Pig latin: A not-so-foreign language for data processing. In: Proceedings of the ACM SIGMOD International Conference on Management of Data; 2008; Vancouver, Canada.
10. Hausenblas M, Nadeau J. Apache drill: interactive ad-hoc analysis at scale. *Big Data*. 2013;1(2):100-104.
11. Presto. 2018. https://prestodb.io/
12. Choi H, Son J, Yang H, et al. Tajo: A distributed data warehouse system on large clusters. Paper presented at: 2013 IEEE 29th International Conference on Data Engineering (ICDE); 2013; Brisbane, Australia.
13. Chang L, Wang Z, Ma T, et al. HAWQ: A massively parallel processing SQL engine in hadoop. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data; 2014; Snowbird, UT.
14. IBM Big SQL. 2018. www.ibm.com/analytics/us/en/technology/big-sql/
15. Apache Phoenix. 2018. http://phoenix.apache.org/
16. Alsubaiee S, Altowim Y, Altwaijry H, et al. Asterixdb: a scalable, open source BDMS. *Proc VLDB Endow*. 2014;7(14):1905-1916.
17. Vertica. 2018. https://www.vertica.com/
18. Amazon Athena. 2018. https://aws.amazon.com/athena/
19. Floratou A, Minhas UF, Özcan F. SQL-on-hadoop: full circle back to shared-nothing database architectures. *Proc VLDB Endow*. 2014;7(12):1295-1306.
20. van Wouw S, Viña J, Iosup A, Epema DickHJ. An empirical performance evaluation of distributed SQL query engines. In: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering; 2015; Austin, TX.
21. Chen Y, Qin X, Bian H, et al. A study of SQL-on-Hadoop systems. In: *Big Data Benchmarks, Performance Optimization, and Emerging Hardware: 4th and 5th Workshops BPOE 2014, Salt Lake City, USA, March 1, 2014 and Hangzhou, China, September 5, 2014, Revised Selected Papers*. Cham, Switzerland: Springer; 2014:154-166.
22. Costea A, Ionescu A, Răducanu B, et al. VectorH: Taking SQL-on-Hadoop to the next level. In: Proceedings of the 2016 International Conference on Management of Data; 2016; San Francisco, CA.
23. Ghazal A, Rabl T, Hu M, et al. BigBench: Towards an industry standard benchmark for big data analytics. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data; 2013; New York, NY.
24. Ivanov T, Beer M. Evaluating Hive and Spark SQL with BigBench. 2015. http://arxiv.org/abs/1512.08417
25. Ivanov T, Beer M. Performance evaluation of spark SQL using BigBench. In: *Big Data Benchmarking: 6th International Workshop, WBDB 2015, Toronto, ON, Canada, June 16-17, 2015 and 7th International Workshop, WBDB 2015, New Delhi, India, December 14-15, 2015, Revised Selected Papers*. Cham, Switzerland: Springer; 2015:96-116.
26. Ghazal A, Ivanov T, Kostamaa P, et al. BigBench V2: The new and improved BigBench. Paper presented at: 2017 IEEE 33rd International Conference on Data Engineering (ICDE); 2017; San Diego, CA.
27. Ivanov T, Taafe J. Exploratory analysis of spark structured streaming. Paper presented at: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering; 2018; Berlin, Germany.
28. Ivanov T, Singhal R. Abench: Big data architecture stack benchmark. Paper presented at: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering; 2018; Berlin, Germany.
29. Ivanov T, Bedué P, Ghazal A, Zicari RV. Adding velocity to BigBench. In: Proceedings of the Workshop on Testing Database Systems; 2018; Houston, TX.
30. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: cluster computing with working sets. *HotCloud*. 2010;10(10-10):95.
31. Apache ORC. 2018. https://orc.apache.org/
32. zlib Home Site. 2018. https://zlib.net/
33. Snappy. 2018. https://google.github.io/snappy/
34. Braams B. *Predicate Pushdown in Parquet and Databricks Spark* [*Master Thesis*]. Amsterdam, The Netherlands: Universiteit van Amsterdam; 2018.
35. He Y, Lee R, Huai Y, et al. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. Paper presented at: 2011 IEEE 27th International Conference on Data Engineering; 2011; Hannover, Germany.
36. Protocol Buffers. 2018. https://developers.google.com/protocolbuffers/
37. Apache Thrift. 2018. https://thrift.apache.org
38. Abadi D, Babu S, Özcan F, Pandis I. SQL-on-hadoop systems: tutorial. *Proc VLDB Endow*. 2015;8(12):2050-2051.
39. Poggi N, Berral JL, Fenech T, et al. The state of SQL-on-Hadoop in the cloud. Paper presented at: 2016 IEEE International Conference on Big Data (Big Data); 2016; Washington, DC.
40. Poggi N, Montero A, Carrera D. Characterizing bigbench queries, hive, and spark in multi-cloud environments. In: *Performance Evaluation and Benchmarking for the Analytics Era: 9th TPC Technology Conference, TPCTC 2017, Munich, Germany, August 28, 2017, Revised Selected Papers*. Cham, Switzerland: Springer; 2017:55-74.
41. Pirzadeh P, Carey MJ, Westmann T. A performance study of big data analytics platforms. Paper presented at: 2017 IEEE International Conference on Big Data (Big Data); 2017; Boston, MA.
42. TPC-H. 2018. www.tpc.org/tpch/
43. TPC-DS. 2018. www.tpc.org/tpcds/
44. Bian H, Yan Y, Tao W, et al. Wide table layout optimization based on column ordering and duplication. In: Proceedings of the 2017 ACM International Conference on Management of Data; 2017; Chicago, IL.
45. Sun L, Franklin MJ, Krishnan S, Xin RS. Fine-grained partitioning for aggressive data skipping. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data; 2014; Snowbird, UT.
46. Sun L, Krishnan S, Xin RS, Franklin MJ. A partitioning framework for aggressive data skipping. *Proc VLDB Endow*. 2014;7(13):1617-1620.
47. Sun L, Franklin MJ, Wang J, Wu E. Skipping-oriented partitioning for columnar layouts. *Proc VLDB Endow*. 2016;10(4):421-432.

48. Apache Arrow. 2018. https://arrow.apache.org/
49. Trivedi A, Stuedi P, Pfefferle J, Schuepbach A, Metzler B. Albis: High-performance file format for big data systems. Paper presented at: 2018 USENIX Annual Technical Conference; 2018; Boston, MA.
50. TPCX-BB. 2018. www.tpc.org/tpcx-bb
51. Baru CK, Bhandarkar MA, Curino C, et al. Discussion of BigBench: A proposed industry standard performance benchmark for big data. In: *Performance Characterization and Benchmarking. Traditional to Big Data: 6th TPC Technology Conference, TPCTC 2014, Hangzhou, China, September 1-5, 2014. Revised Selected Papers*. Cham, Switzerland: Springer; 2014:44-63.
52. Chowdhury B, Rabl T, Saadatpanah P, Du J, Jacobsen H. A BigBench implementation in the Hadoop ecosystem. In: *Advancing Big Data Benchmarks: Proceedings of the 2013 Workshop Series on Big Data Benchmarking, WBDB.cn, Xi'an, China, July 16-17, 2013 and WBDB.us, San José, CA, USA, October 9-10, 2013, Revised Selected Papers*. Cham, Switzerland: Springer; 2013:3-18.
53. Frankfurt Big Data Lab: Big-Bench-Setup. 2018. https://github.com/BigData-Lab-Frankfurt/Big-Bench-Setup
54. Manyika J, Chui M, Brown B, et al. Big Data: The Next Frontier for Innovation, Competition, and Productivity. 2011. http://www.mckinsey.com/Insights/MGI/Research/Technology_and_Innovation/Big_data_The_next_frontier_for_innovation
55. TPC - Current Specifications - tpcx-bb v1.2.0. 2019. http://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPCX-BB_v1.2.0.pdf
56. Apache Hadoop. 2018. http://hadoop.apache.org/
57. Apache Hive. 2018. https://hive.apache.org/
58. Apache Mahout. 2018. http://mahout.apache.org/
59. Apache OpennLP. 2018. https://opennlp.apache.org/
60. SparkSQL Configuration. 2018. https://spark.apache.org/docs/latest/configuration.html#spark-sql
61. Blue R. Parquet Performance Tuning: The Missing Guide. 2016. https://conferences.oreilly.com/strata/strata-ny-2016/public/schedule/detail/52110
62. Serde - Apache Hive. 2018. https://cwiki.apache.org/confluence/display/Hive/SerDe
63. Hive Configuration Properties. 2018. https://cwiki.apache.org/confluence/display/Hive/Configuration+Properties
64. Rabl T, Frank M, Sergieh HM, Kosch H. A data generator for cloud-scale benchmarking. In: *Performance Evaluation, Measurement and Characterization of Complex Systems: Second TPC Technology Conference, TPCTC 2010, Singapore, September 13-17, 2010, Revised Selected Papers*. Berlin, Germany: Springer; 2010:41-56.
65. Frankfurt Big Data Lab: Detailed-Results. 2018. https://github.com/BigData-Lab-Frankfurt/ColumnarFileFormatsEvaluation
66. Spark-Internals. 2018. https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-DAGScheduler-Stage.html
67. Intel PAT Tool. 2018. https://github.com/intel-hadoop/PAT
68. Hartigan JA, Wong MA. Algorithm AS 136: a k-means clustering algorithm. *J Royal Stat Soc Ser C (Appl Stat)*. 1979;28(1):100-108.
69. Spark SQL and DataFrames - Spark 2.3.0 Documentation. 2018. http://spark.apache.org/sql/
70. Femminella M, Pergolesi M, Reali G. Performance evaluation of edge cloud computing system for big data applications. Paper presented at: 2016 5th IEEE International Conference on Cloud Networking (Cloudnet); 2016; Pisa, Italy.
71. Hue. 2018. http://gethue.com/
72. Baruffa G, Femminella M, Pergolesi M, Reali G. A Big Data architecture for spectrum monitoring in cognitive radio applications. *Ann Telecommun*. 2018;73(7-8):451-461.
73. Rajendran S, Calvo-Palomino R, Fuchs M, et al. Electrosense: Open and big spectrum data. *IEEE Commun Mag*. January 2018;56(1):210-217.

How to cite this article: Ivanov T, Pergolesi M. The impact of columnar file formats on SQL-on-Hadoop engine performance: A study on ORC and Parquet. *Concurrency Computat Pract Exper*. 2019;e5523. https://doi.org/10.1002/cpe.5523

APPENDIX A

BigBench Q08 (MapReduce/Python)

```
For online sales, compare the total sales monetary amount in which customers checked online reviews
3 days before making the purchase and that of sales in which customers did not read reviews.
Consider only online sales for a specific category in a given year.

-- Resources
ADD FILE ${hiveconf:QUERY_DIR}/q08_filter_sales_with_reviews_viewed_before.py;
DROP TABLE IF EXISTS ${hiveconf:TEMP_TABLE2};
DROP TABLE IF EXISTS ${hiveconf:TEMP_TABLE3};
DROP TABLE IF EXISTS ${hiveconf:TEMP_TABLE1};

--DateFilter
CREATE TABLE ${hiveconf:TEMP_TABLE1} AS
SELECT d_date_sk
FROM date_dim d
WHERE d.d_date >= '${hiveconf:q08_startDate}'
AND   d.d_date <= '${hiveconf:q08_endDate}';

--PART 1 --
CREATE TABLE ${hiveconf:TEMP_TABLE2} AS
SELECT DISTINCT wcs_sales_sk
FROM (
  FROM (
    SELECT  wcs_user_sk,
            (wcs_click_date_sk * 86400L + wcs_click_time_sk) AS tstamp_inSec,
            wcs_sales_sk,
            wp_type
    FROM web_clickstreams
    LEFT SEMI JOIN ${hiveconf:TEMP_TABLE1} date_filter
    ON (wcs_click_date_sk = date_filter.d_date_sk and wcs_user_sk IS NOT NULL)
    JOIN web_page w ON wcs_web_page_sk = w.wp_web_page_sk
    DISTRIBUTE BY wcs_user_sk
    SORT BY wcs_user_sk,tstamp_inSec,wcs_sales_sk,wp_type
  ) q08_map_output
  -- input: web_clicks in a given year
  REDUCE  wcs_user_sk,
          tstamp_inSec,
          wcs_sales_sk,
          wp_type
  USING 'python q08_filter_sales_with_reviews_viewed_before.py
                review ${hiveconf:q08_seconds_before_purchase}'
  AS (wcs_sales_sk BIGINT)
) sales_which_read_reviews;

--PART 2 --
CREATE TABLE IF NOT EXISTS ${hiveconf:TEMP_TABLE3} AS
SELECT ws_net_paid, ws_order_number
FROM web_sales ws
JOIN ${hiveconf:TEMP_TABLE1} d ON ( ws.ws_sold_date_sk = d.d_date_sk);

--PART 3
--CREATE RESULT TABLE. Store query result externally in output_dir/qXXresult/
DROP TABLE IF EXISTS ${hiveconf:RESULT_TABLE};
CREATE TABLE ${hiveconf:RESULT_TABLE} (
  q08_review_sales_amount    BIGINT,
  no_q08_review_sales_amount BIGINT
)
```

```
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n'
STORED AS ${env:BIG_BENCH_hive_default_fileformat_result_table} LOCATION '${hiveconf:RESULT_DIR}';

-- the real query part--
INSERT INTO TABLE ${hiveconf:RESULT_TABLE}
SELECT
  q08_review_sales.amount AS q08_review_sales_amount,
  q08_all_sales.amount - q08_review_sales.amount AS no_q08_review_sales_amount
FROM (
  SELECT 1 AS id, SUM(ws_net_paid) as amount
  FROM ${hiveconf:TEMP_TABLE3} allSalesInYear
  LEFT SEMI JOIN ${hiveconf:TEMP_TABLE2} salesWithViewedReviews
  ON allSalesInYear.ws_order_number = salesWithViewedReviews.wcs_sales_sk
) q08_review_sales
JOIN (
  SELECT 1 AS id, SUM(ws_net_paid) as amount
  FROM ${hiveconf:TEMP_TABLE3} allSalesInYear
) q08_all_sales
ON q08_review_sales.id = q08_all_sales.id;

--cleanup--
DROP TABLE IF EXISTS ${hiveconf:TEMP_TABLE2};
DROP TABLE IF EXISTS ${hiveconf:TEMP_TABLE3};
DROP TABLE IF EXISTS ${hiveconf:TEMP_TABLE1};
```

APPENDIX B

BigBench Q10 (HiveQL/OpenNLP)

```
For all products, extract sentences from its product reviews that contain positive or
negative sentiment and display for each item the sentiment polarity of the extracted
sentences (POS OR NEG) and the sentence and word in sentence leading to this classification.

-- Resources
ADD JAR ${env:BIG_BENCH_QUERIES_DIR}/Resources/opennlp-maxent-3.0.3.jar;
ADD JAR ${env:BIG_BENCH_QUERIES_DIR}/Resources/opennlp-tools-1.6.0.jar;
ADD JAR ${env:BIG_BENCH_QUERIES_DIR}/Resources/bigbenchqueriesmr.jar;
CREATE TEMPORARY FUNCTION extract_sentiment AS 'io.bigdatabenchmark.v1.queries.q10.SentimentUDF';

-- CREATE RESULT TABLE. Store query result externally in output_dir/qXXresult/
DROP TABLE IF EXISTS ${hiveconf:RESULT_TABLE};
CREATE TABLE ${hiveconf:RESULT_TABLE} (
  item_sk         BIGINT,
  review_sentence STRING,
  sentiment       STRING,
  sentiment_word  STRING
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n'
STORED AS ${env:BIG_BENCH_hive_default_fileformat_result_table} LOCATION '${hiveconf:RESULT_DIR}';

-- the real query part
INSERT INTO TABLE ${hiveconf:RESULT_TABLE}
SELECT item_sk, review_sentence, sentiment, sentiment_word
FROM (
  SELECT extract_sentiment(pr_item_sk, pr_review_content)
  AS (item_sk, review_sentence, sentiment, sentiment_word)
  FROM product_reviews
) extracted
ORDER BY item_sk,review_sentence,sentiment,sentiment_word;
```

APPENDIX C

BigBench Q12 (Pure HiveQL)

```
Find all customers who viewed items of a given category on the web in a given month
and year that was followed by an in-store purchase of an item from the same category
in the three consecutive months.

DROP TABLE IF EXISTS ${hiveconf:RESULT_TABLE};
CREATE TABLE ${hiveconf:RESULT_TABLE} (
  u_id BIGINT
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n'
STORED AS ${env:BIG_BENCH_hive_default_fileformat_result_table} LOCATION '${hiveconf:RESULT_DIR}';

INSERT INTO TABLE ${hiveconf:RESULT_TABLE}
SELECT DISTINCT wcs_user_sk -- Find all customers
FROM
( -- web_clicks viewed items in date range with items from specified categories
  SELECT
    wcs_user_sk,
    wcs_click_date_sk
  FROM web_clickstreams, item
  WHERE wcs_click_date_sk BETWEEN 37134 AND (37134 + 30) -- in a given month and year
  AND i_category IN (${hiveconf:q12_i_category_IN}) -- filter given category
  AND wcs_item_sk = i_item_sk
  AND wcs_user_sk IS NOT NULL
  AND wcs_sales_sk IS NULL --only views, not purchases
) webInRange,
(  -- store sales in date range with items from specified categories
  SELECT
    ss_customer_sk,
    ss_sold_date_sk
  FROM store_sales, item
  WHERE ss_sold_date_sk BETWEEN 37134 AND (37134 + 90) -- in the three consecutive months.
  AND i_category IN (${hiveconf:q12_i_category_IN}) -- filter given category
  AND ss_item_sk = i_item_sk
  AND ss_customer_sk IS NOT NULL
) storeInRange -- join web and store
WHERE wcs_user_sk = ss_customer_sk
AND wcs_click_date_sk < ss_sold_date_sk -- buy AFTER viewed on website
ORDER BY wcs_user_sk;
```

APPENDIX D

BigBench Q25 (HiveQL/SparkMLlib)

```
Customer segmentation analysis: Customers are separated along the  following key shopping
dimensions: recency of last visit, frequency of visits and monetary amount. Use the store
and online purchase data during a given year to compute. After model of separation is build,
report for the analysed customers to which "group" they where assigned.

DROP TABLE IF EXISTS ${hiveconf:TEMP_TABLE};
CREATE TABLE ${hiveconf:TEMP_TABLE} (
  cid               BIGINT,
  frequency         BIGINT,
  most_recent_date  BIGINT,
  amount            decimal(15,2)
);

-- Add store sales data
INSERT INTO TABLE ${hiveconf:TEMP_TABLE}
SELECT
  ss_customer_sk                    AS cid,
  count(distinct ss_ticket_number)  AS frequency,
  max(ss_sold_date_sk)              AS most_recent_date,
  SUM(ss_net_paid)                  AS amount
FROM store_sales ss
JOIN date_dim d ON ss.ss_sold_date_sk = d.d_date_sk
WHERE d.d_date > '${hiveconf:q25_date}'
AND ss_customer_sk IS NOT NULL
GROUP BY ss_customer_sk;

-- Add web sales data
INSERT INTO TABLE ${hiveconf:TEMP_TABLE}
SELECT
  ws_bill_customer_sk             AS cid,
  count(distinct ws_order_number) AS frequency,
  max(ws_sold_date_sk)            AS most_recent_date,
  SUM(ws_net_paid)                AS amount
FROM web_sales ws
JOIN date_dim d ON ws.ws_sold_date_sk = d.d_date_sk
WHERE d.d_date > '${hiveconf:q25_date}'
AND ws_bill_customer_sk IS NOT NULL
GROUP BY ws_bill_customer_sk;

--ML-algorithms expect double values as input for their Vectors.
DROP TABLE IF EXISTS ${hiveconf:TEMP_RESULT_TABLE};
CREATE TABLE ${hiveconf:TEMP_RESULT_TABLE} (
  cid        BIGINT,--used as "label", all following values are used as Vector for ML-algorithm
  recency    double,
  frequency  double,
  totalspend double
);

INSERT INTO TABLE ${hiveconf:TEMP_RESULT_TABLE}
SELECT
  cid            AS cid,
  CASE WHEN 37621 - max(most_recent_date) < 60 THEN 1.0 ELSE 0.0 END
                 AS recency, -- 37621 == 2003-01-02
  SUM(frequency) AS frequency, --total frequency
  SUM(amount)    AS totalspend --total amount
FROM ${hiveconf:TEMP_TABLE}
GROUP BY cid
ORDER BY cid;

--- CLEANUP--
DROP TABLE ${hiveconf:TEMP_TABLE};
```