Cluster Computing
https://doi.org/10.1007/s10586-022-03718-9

On the performance of SQL scalable systems on Kubernetes: a comparative study

Cristian Cardas[1] · José F. Aldana-Martín[1] · Antonio M. Burgueño-Romero[1] · Antonio J. Nebro[1,2] · Jose M. Mateos[3] · Juan J. Sánchez[3]

Received: 21 June 2022 / Revised: 30 July 2022 / Accepted: 11 August 2022

Abstract

The popularization of Hadoop as the the-facto standard platform for data analytics in the context of Big Data applications has led to the upsurge of SQL-on-Hadoop systems, which provide scalable query execution engines allowing the use of SQL queries on data stored in HDFS. In this context, Kubernetes appears as the leading choice to simplify the deployment and scaling of containerized applications; however, there is a lack of studies about the performance of SQL-on-Hadoop systems deployed on Kubernetes, and this is the gap we intend to fill in this paper. We present an experimental study involving four representative SQL scalable platforms: Apache Drill, Apache Hive, Apache Spark SQL and Trino. Concretely, we analyze the performance of these systems when they are deployed on a Hadoop cluster with Kubernetes by using the TPC-H benchmark. The results of our study can help practitioners and users about what they can expect in terms of performance if they plan to use the advantages of Kubernetes to deploy applications using the analyzed SQL scalable platforms.

Keywords Scalable SQL systems · Hadoop · Kubernetes · Apache Spark · Trino · Apache Drill · Hive MR3

1 Introduction

In the last years, Hadoop has become the standard platform for Big Data applications by providing a cost-effective, open-source, and scalable solution [1]. Core components of Hadoop architecture are HDFS (Hadoop Distributed File System) and Yarn (Yet Another Resources Negotiator); the former allows to store large volumes of data, and the latter provides resource management and scheduling to run jobs. Hadoop clusters do not require specific hardware, and they can be installed on-premise or on-cloud by using providers such as AWS or Azure.

Although Hadoop can store both structured and unstructured data in HDFS, many users prefer to use SQL as the base language of data analytics applications to query structured data sources. As a consequence, some SQL-on-Hadoop systems have emerged, such as Apache Hive [2], Apache Impala [3], Trino [4] (formerly known as Presto), Apache Drill [5], or Apache Spark-SQL [6]. These systems are built on top of scalable query engines able to exploit the features of Hadoop to execute SQL queries on large amounts of data effectively.

✉ Antonio J. Nebro
ajnebro@uma.es

Cristian Cardas
criscardas@uma.es

José F. Aldana-Martín
jfaldanam@uma.es

Antonio M. Burgueño-Romero
ambrbr@uma.es

Jose M. Mateos
jose.manuel.mateos@ericsson.com

Juan J. Sánchez
juan.jesus.sanchez.sanchez@ericsson.com

[1] ITIS Software, University of Málaga, Ada Byron Research Building, 29071 Málaga, Málaga, Spain

[2] Departamento de Lenguajes y Ciencias de la Computación, University of Málaga, E.T.S. de Ingeniería Informática, 29071 Málaga, Málaga, Spain

[3] Ericsson, Andalusia Technology Park, C/Severo Ochoa 55-Building 2, 29590 Málaga, Spain

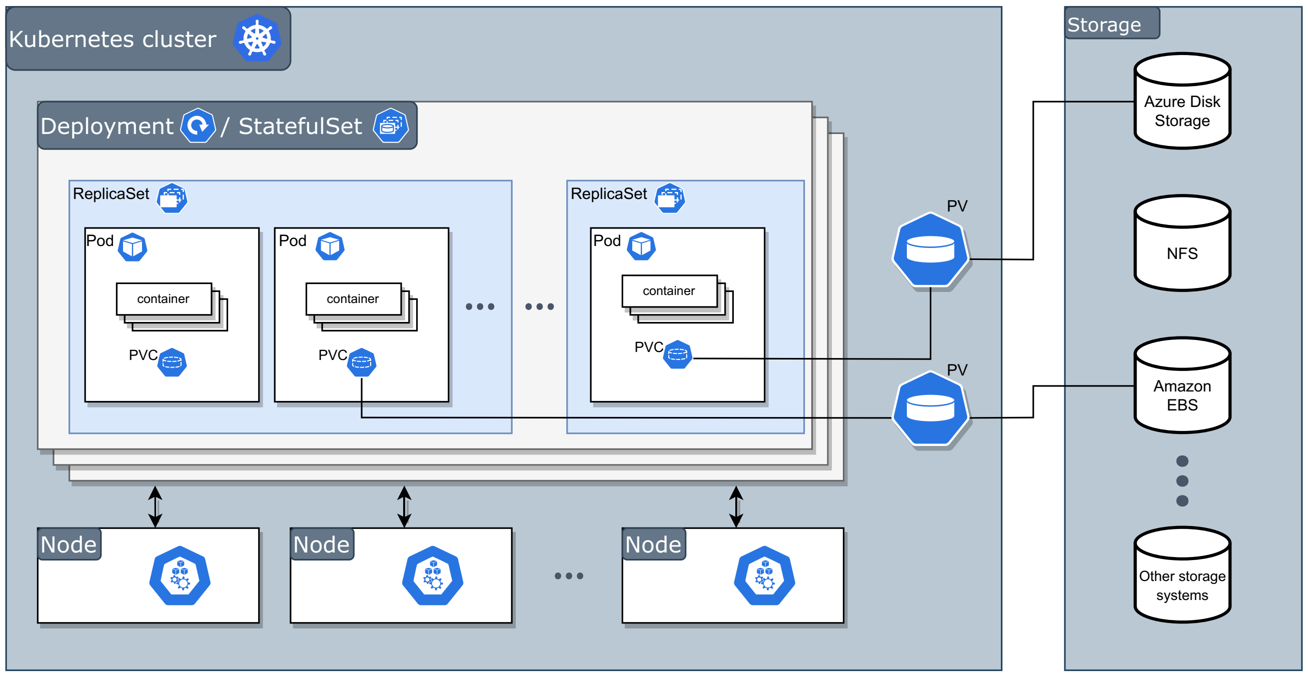


Fig. 1 Kubernetes architecture

In this context, the deployment, scaling, and management of applications on Hadoop clusters are significant issues that have to be addressed. One of the most widely used systems for coping with these tasks is Kubernetes [7], an open-source platform for managing containerized workloads, services, and applications. Originally developed by Google, Kubernetes' features include service discovery, load balancing, horizontal scaling, storage orchestration, and self-healing.

Analyzing the performance of SQL scalable systems deployed on Kubernetes is a study that, as far as we are concerned, has not been done so far, so in this work, we intend to fill this gap. This paper focuses on evaluating the performance of four representative SQL-on-Hadoop systems (namely Drill, Hive, Spark SQL, and Trino) by deploying them in a cluster on Kubernetes. Our main goal is to assess the systems' performance on Kubernetes with the TPC-H benchmark. We are also interested in investigating the potential limitations or difficulties of adapting the selected systems on Kubernetes.

The main contributions of this paper are summarized as follows:

- A review of the literature for SQL scalable systems and their performance in query execution in Big Data ecosystems.
- An analysis of the requirements, limitations, and problems encountered in deploying the selected SQL engines on Kubernetes.
- Definition of a clean experimentation methodology to ease the reproducibility of the reported results.
- A comparison between the systems and the evaluation of the impact in performance from working with Kubernetes using the TPC-H benchmark.
- A repository hosted in GitHub containing the adapted TPC-H queries for the four analyzed systems and scripts for the automatic deployment and execution on Kubernetes.

The rest of the paper is organized as follows: Sect. 2 includes works related to the main topics of this paper, including comparative studies on scalable SQL-on-Hadoop systems and the use of Kubernetes on similar systems. Section 3 provides an in-depth look at different technologies covered in this work, such as the structure of Kubernetes, the auto-scaling capabilities of Kubernetes or the characteristics of each SQL system used. Section 4 contains the methodology, experimental setup, experiments, analyses, obtained results, and evaluation. To conclude the paper, Sect. 5 offers some conclusions on these topics and interesting lines for further research.

2 Related work

The interest in scalable SQL-on-Hadoop systems has led to research works to benchmark and compare them. One of the first studies was presented in [8], which compared Hive and Impala on the TPC-H [9] and TPC-DS [10] benchmarks on a 21 node cluster summing up 240 processing cores. Also, in 2014, [11] analyzed the performance of Hive, Impala, Presto, and two nowadays extinguished systems, Stinger and Shark, on clusters composed of up to 100 nodes and using some queries and datasets taken from the TPC-DS benchmark. In [12], the systems Impala, Spark SQL, and Drill were analyzed and compared by using a cluster of virtual machines deployed on Amazon EC2; two benchmarks were used, WDA [13] and TPC-H. The study presented in [14] includes Drill, HAWQ [11], Hive, Impala, Presto, and Spark; the benchmark is TPC-H, and the cluster contains four worker nodes having eight cores each. TPC-H and TPD-DS plus TPCx-BB[1] are the subjects of analysis in [15], who used them to compare Hive, Spark SQL, Impala, and Presto on a cluster of five nodes with 16 cores, 64 GB of memory, and 1.2 TB of SSD disk each.

All these studies perform similar analyses in the sense that the target systems many of SQL scalable systems tested on Hadoop clusters by using some TPC benchmarks. However, none of them considers the possibility of deploying them using Kubernetes. In this sense, we can mention the work of C. Zhu, B. Han, and Y. Zhao, which is focused on comparing Spark on the bare metal and Kubernetes [16]; however, this study considers a set of four Spark applications (word count, sort, k-means, SQL join) and it does not focus on a deep analysis of the Spark SQL engine. A similar case can be found in [17], which presents a study of four Spark schedulers, including Mesos, Kubernetes, Yarn, and stand-alone.

3 Technologies

This section describes the technologies covered in the paper: Kubernetes and the SQL-on-Hadoop systems Drill, Hive, Spark SQL and Trino.

3.1 Kubernetes

Kubernetes (https://kubernetes.io.) is an open-source platform that provides mechanisms for deploying, maintaining, scaling, and healing containerized applications. It does so while offering high availability and orchestration capabilities, controlled by the Kubelet service, perfect for production-ready applications, where minimizing computing resources and reassuring its uptime is crucial. In this sense, migrating the systems considered in this work to Kubernetes brings a means for automated scaling, limiting resource consumption, fault tolerance, and an efficient way of deploying and configuring the platform.

Some key concepts and elements related to Kubernetes are enumerated next:

- *Nodes* are the physical or virtual machines where everything will run and is controlled by the Kubelet process. By default, there are two types of nodes, a master and the workers. The master contains by default a taint (Kubernetes term for a "mark") that disallows most deployments to run on it, to guarantee that it has enough free resources to fulfill its duties as the master node.
- *Pods* are the minor deployable units of computing in Kubernetes. They are composed of one or more containers, running in one of the supported container runtime environments like Docker or directly on containerd, with shared storage and network resources and context. The workers and master nodes' containers are defined as pods by other workload resources.
- *ReplicaSets* are in charge of maintaining a stable set of *Pods* running at any time, using replicas to guarantee high availability. They create and delete Pods to reach the desired number with the help of a pod template.
- *Deployments* are a high-level concept that manages and provides declarative updates for *Pods* and *ReplicaSets* at a controlled rate.
- *StatefulSets* manage the deployment and scaling of a set of Pods, guaranteeing their ordering and uniqueness and maintaining their identifier across any rescheduling. This is useful for having unique network identifiers, graceful deployment, and stable and scaling persistent storage.
- *Persistent Volumes* and *Persistent Volume Claims* (PV and PVC respectively) define the dynamic for storage management in Kubernetes. A PV is a piece of storage that can be manually or dynamically provisioned by defining a *StorageClass*. PVs are regular volumes that have a lifecycle independent of any Pod bound. PVs can be supplied from many sources, including but not limited to NFS, Azure Disk, or Amazon EBS. Meanwhile, PVCs are requests for storage by users, which can specify size and access modes.

Figure 3.1 shows all the elements enumerated above and relates them in a diagram that represents the general structure of Kubernetes.

An application deployment on Kubernetes is defined through YAML manifests called "charts", which are collections of versioned, pre-configured application resources as a single unit. In order to make chart installation's transparent for users, there exist package managers like

[1] http://www.tpc.org/tpcx-bb.

Helm[2] which we have used for the deployment of the SQL systems introduced in the following section.

Deploying scalable SQL systems on Kubernetes brings the additional benefit of being able to allow Kubernetes to handle the auto-scaling of the cluster according to the load they are under at any given time. It allows the systems to consume little resources when the SQL system is idle but scale up rapidly and automatically whenever the load is increased, for example, when a group of users use the system concurrently.

This feature provided by Kubernetes can be a powerful cost-saving method, especially when deploying the SQL systems on the cloud, where you are billed by the resources you are consuming at any given time. By setting a target resource usage percentage, e.g.: aiming to keep the nodes at 75% load, Kubernetes adds or subtracts pods to the deployment of the SQL system. By increasing or decreasing this target usage, the scaling will be done more or less aggressively, providing more flexibility for different use cases.

As an example use case to benefit from this feature, we can imagine a system where users from different time zones query the underlying SQL scalable system. This event causes the load on the SQL system to spike when the users of the most populated time zones are active while having lower loads close to inactivity during the rest of the hours. In that context, Kubernetes would automatically scale the number of pods on the SQL system to handle the load at any given time.

3.2 SQL scalable systems

Four SQL scalable systems have been selected as the most relevant from reviewing related works, namely Trino, Hive, Spark, and Drill. Next, a brief description of them is given:

- Trino (https://trino.io/) is an Apache 2.0 licensed, distributed SQL query engine, which was forked from the original Presto[3] project. It was designed from the ground up for fast queries against any amount of data. It supports any data source, including relational and non-relational sources, via its connector architecture.
- Apache Hive (https://hive.apache.org) is an open-source project that provides a SQL-like query engine. It is developed in Java and facilitates reading, writing, and managing large datasets residing in HDFS storage. The framework can be projected onto the stored data, providing a command-line tool and a JDBC driver to connect users to Hive.
- Apache Spark (http://spark.apache.org) allows query execution using a subset of SQL on a variety of data sources. Its scalable processing engine, is particulary suitable if the resulting data is analyzed using Spark's data frames or machine learning APIs after performing SQL queries.
- Apache Drill (https://drill.apache.org/) is a SQL query engine designed to analyze data stored in Hadoop and non-relational databases, including MongoDB and Hbase. It can scale from laptops to 1000-node clusters and has symmetric architecture where all nodes are equal.

3.3 Deployment on Kubernetes

Some of the previously introduced SQL systems were not initially designed for Kubernetes. However, except for Hive, they have adopted support for it. Due to the nature of these complex deployments, most enterprises and organizations decide on their use on the cloud, avoiding managing such systems' deployment in local clusters. Therefore, although they exist, the official guidelines for deploying systems on Kubernetes are still incomplete, a work in progress or lacking in information, and the community around them is rather small. Some organizations and other third parties also maintain non-official guidelines for the deployments, often used internally and leading to some of them being better maintained than official ones. These Helm charts are shared in repositories like artifacthub.io[4].

A summary of the selected charts is included in Table 1. The Spark and Trino charts are well maintained, allowing both internal and Kubernetes resource configurations to be easily made. The chart for Drill, albeit official, has not been updated to the latest versions of Drill and presents some issues that caused the system's configuration a demanding task, as it does not follow the standard good practices, i.e., defining Kubernetes resource limits. As far as we are concerned, this is the only publicly available deployment for Drill on Kubernetes. We did not find any operational deployment of Apache Hive on Kubernetes. However, starting from 2018, DataMonad released its engine called MR3 (MapReduce 3), which allows Hive to be run on Kubernetes[5]. This deployment is extraordinarily complex and required several weeks of investigation as the official MR3 documentation offered little detail on its deployment. We did find several issues regarding their repositories and updates, which further complicated our experience. The number of parameters and files to modify was

[2] https://helm.sh/.

[3] https://prestodb.io

[4] https://artifacthub.io/.

[5] https://www.datamonad.com/.

Table 1 Summary information on the selected SQL engines

	Trino	Hive on MR3	Spark	Drill
SQL support	ANSI SQL compliant	HiveQL	ANSI SQL compliant	ANSI SQL compliant
License	Apache 2.0	Apache 2.0	Apache 2.0	Apache 2.0
Initial release	2019	2018	2014	2012
Current release	363, October 2021	1.3, August 2021	3.1.2, June 2021	1.17.0, December 2019

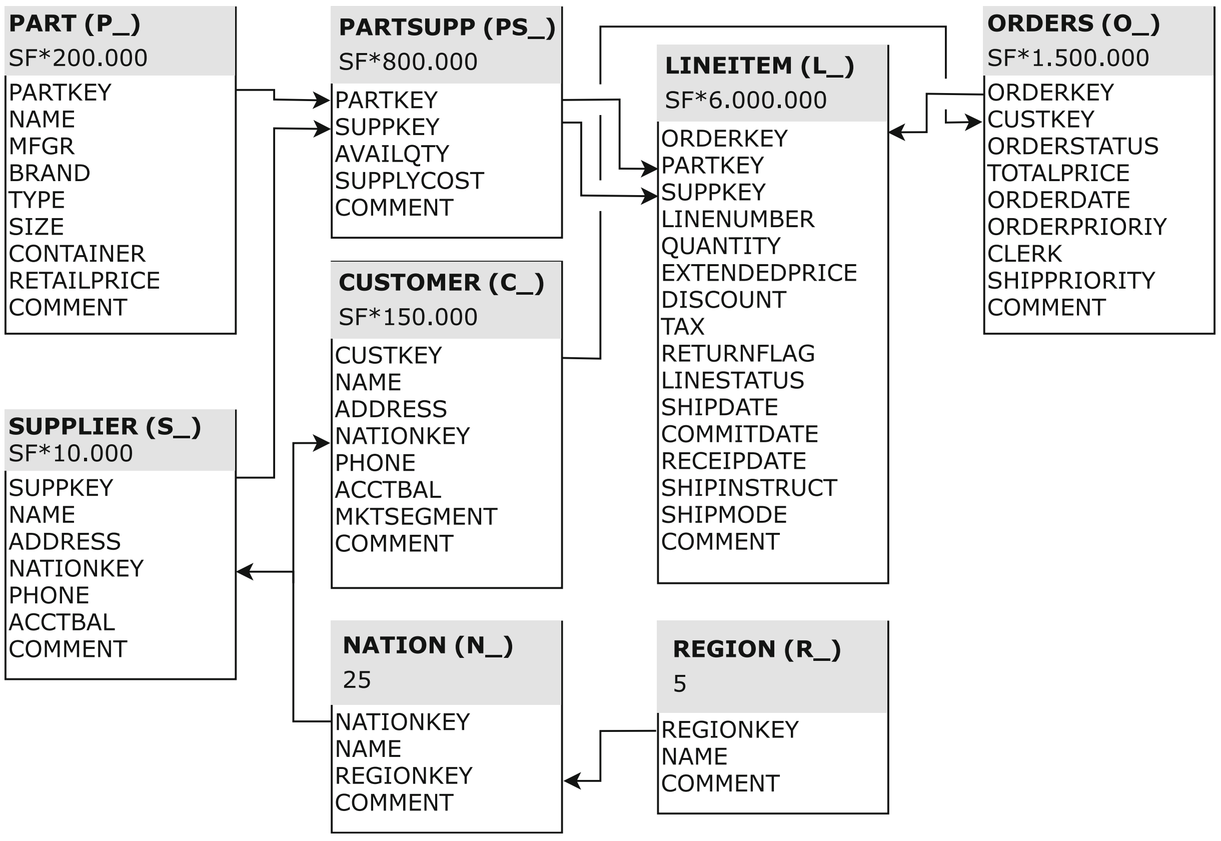


Fig. 2 The TPC-H schema

overwhelming regarding its configuration and fine-tuning. As there were no references or blogs for debugging, which made the task time-consuming.

4 Experimentation

In this section, the selected benchmark, the followed methodology, and the performed experiments are described in detail.

4.1 The TPC-H Benchmark

The Transaction Processing Performance Council[6] (TPC) is a non-profit consortium that defines standard benchmarks for different systems, such as TPC-C, TPC-DS, TPCx-HS, etc. In this work, we will use TPC-H [18].

The TPC-H is a decision support benchmark that consists of a suite of business-oriented ad-hoc queries, whose data model and query workload is complex enough to serve as a reasonable set of analytic tasks. It illustrates decision support systems that examine large volumes of data with

[6] Transaction processing Performance Council: tpc.org.

Table 2 TPC-H queries characterization

Query	Avg	Count	Min/Max	Sum	From	Group by	Oder by	Nested queries
Q1	3	1		4	1	●	●	
Q2			1		5		●	●
Q3				1	3	●	●	
Q4		1		2		●	●	●
Q5				1	6	●	●	
Q6				1	1			
Q7				1	5	●	●	●
Q8				3	7	●	●	●
Q9				1	6	●	●	●
Q10				1	4	●	●	
Q11				3	3	●	●	●
Q12				2	2	●	●	
Q13		1			2	●	●	●
Q14				2	2			
Q15			1		2	●	●	●
Q16		1			3	●	●	●
Q17	1			1	2			●
Q18				2	3	●	●	●
Q19				1	2			
Q20				1	4		●	●
Q21		1			4	●	●	●
Q22		1		1	2	●	●	●

complex queries. The first version was released in 1999 and has been updated throughout the years. For this work, the considered version is 3.0.0 from February 2021.

The DBGen software package from TPC was used to produce the data to populate the database. This package can be downloaded free of charge from the TPC Website. The complete table schema defined in the official TPC-H specification can be observed in Fig. 2, along with each table's columns, rows, their respective foreign keys and their relationship with other tables. It can be seen that tables *lineitem, orders, and partsupp* are the tables with the most number of rows. Note that most tables' size depends on the scale factor constant (SF); this will be discussed in Sect. 4.2.

Table 2 presents a summary of each query's clauses and operations, which facilitates the interpretation of the results. In the table, we can observe several queries that read data from several tables, like Q2, Q5, Q7, Q8, and Q9, making them moderately resource-intensive queries. Most of them present aggregation functions and group by and order by clauses. The eligible substitution parameters for the queries followed the ones defined by Yuntao Jia[7].

[7] https://issues.apache.org/jira/browse/hive-600.

4.2 Methodology

To describe the ability of the different systems to improve subsequently executed queries due to caching or similar optimization techniques, they were executed a total of 5 times per SQL system. Every system was evaluated in an isolated fashion, without external workloads or interferences. For evaluating the system's ability to scale to different volumes of data, three different scale factors (10, 100, and 300 GB) were considered for the benchmark. The data was stored in an external HDFS cluster under the same network in TBL format, partitioned in files of 10 GB. A series of Bash scripts allowed all systems to be independently deployed, ensuring a clean state for each execution. These scripts deploy, define the TPC-H schema accordingly to the specific platform and execute the queries a given number of consecutive times. After each query has been executed, the schema is discarded, and the cluster's memory is cleansed to avoid caching. The system's output is written to files, then parsed to extract the execution times for visualization.

For measuring the performance of Trino, its Hive Metastore connector has been used, along with the deployment of Hive, where the TPC-H tables are created. The same procedure with the Metastore was followed for

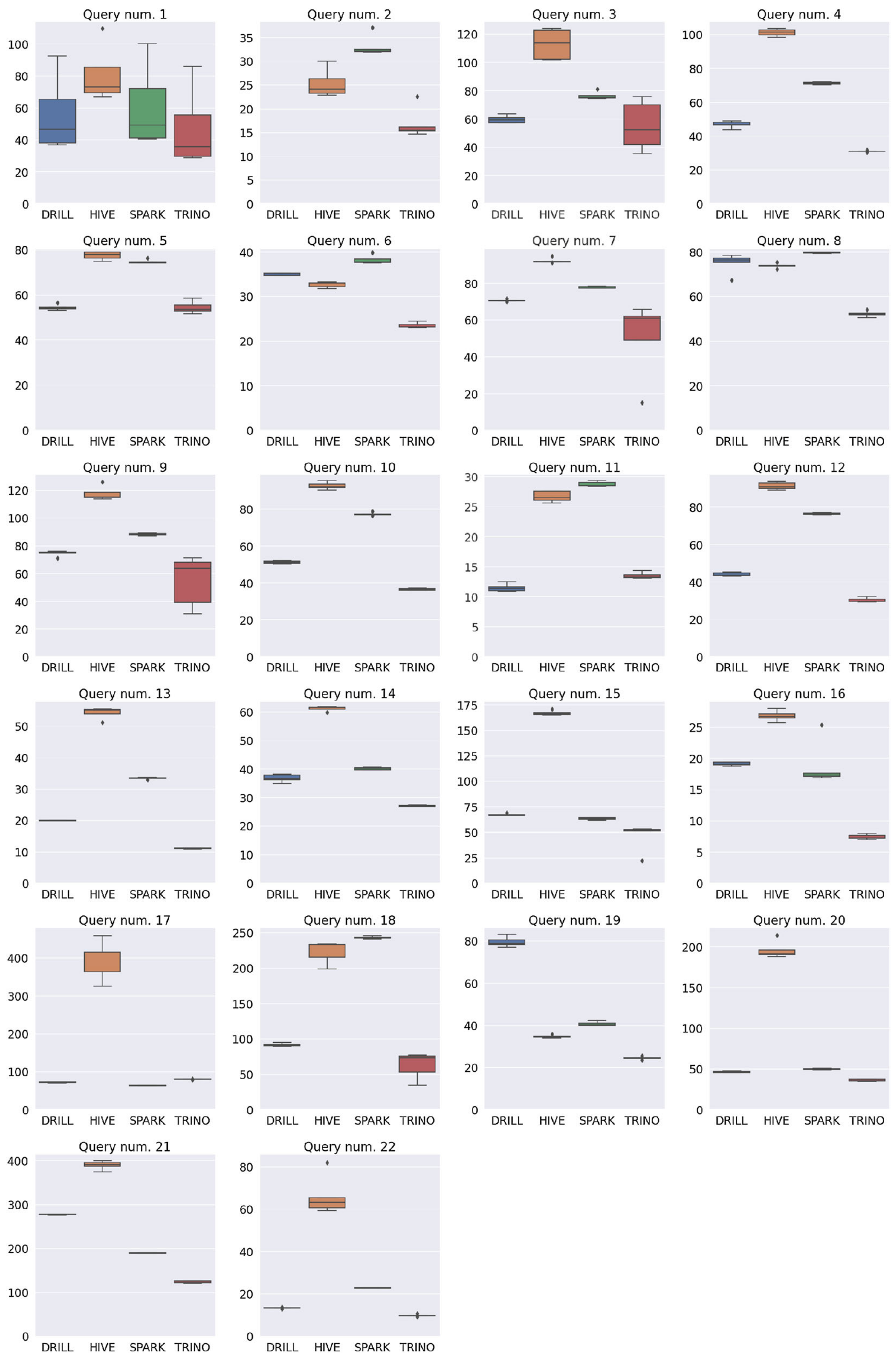


Fig. 3 Boxplots of the computing times obtained when performing the TPC-H queries using a 100 GB scale factor

Hive, using the beeline JDBC driver for querying the tables. For Spark, Python scripts with the Pyspark library were produced. These scripts read and lazily create the tables in memory to execute the queries. Finally, a view for each table was created for Drill to query the data. The schema for each table had to be implicitly declared as the TBL format does not provide column names and types.

For the sake of reproducibility, the developed scripts for automated deployment, TPC-H schema definition, and queries are also available in the GitHub repository of the paper.

4.3 Experimental setup

The system used for the performed experiments is a seven-node Kubernetes cluster with CentOS 7. Each node is a virtual machine provided with 12 virtual Intel Xeon Skylake processor cores at 2.10 GHz and 64 GB of RAM. The data were stored in a separated HDFS cluster, connected to the main Kubernetes cluster through a 10 gigabit network, leaving the Kubernetes nodes solely for processing purposes.

On the Kubernetes cluster, the taint that blocks the scheduling of pods on the master node has been removed, allowing the use of all seven nodes for the SQL system clusters.

The Kubernetes pod configurations for all systems were fine-tuned during the experiments, so that the maximum performance could be extracted from all of them while harnessing the processing capabilities of the hardware. The maximum memory for the pods was set to 62 GB and the number of CPU cores was set to 11 for requests and 12 for limits, leaving 2 GB and 1 core left for OS requirements.

4.4 Performance evaluation

As commented previously, we have consecutively executed each TPC-H query on every platform. In this section, we first analyze the results obtained when using the scale factor of 100 GB; after that, we extend the study to cover the systems' performance including the 10 GB, 100 GB, and 300 GB scale factors to determine whether the observed behaviors of the first analysis hold when using different data sizes.

4.4.1 Results with 100 GB scale factor

The computing times using the 100 GB scale factor are summarized in Fig. 3, which shows a boxplot chart per query. At first glance, we can observe that the dispersion of the times is low in most of the cases except for Query 1. Drill and Spark are very consistent (the boxes are almost plain), while Hive and Trino show significant differences in some particular queries (e.g, queries 3, 7, 9, and 18). Regardless of the scale factor, we did not manage to execute Query 2 due to memory issues on Drill.

A performance ranking would be led by Trino, which is the fastest system for most queries, followed by Drill, Spark, and Hive. Although the number of runs is five, the fact that the boxplots do not overlap (with the exception of Query 1), allows us to claim that the differences are significant.

To have an insight on the quantitative performance of the four systems, we provide the mean and speed-up of the query computing times in Table 3. The speed-ups are computed as the highest execution time divided by the current system computing time [14], so a speed-up of 1.0 in the table means that the system is the slowest in the corresponding query. The best results are highlighted in boldface, and it can be seen that Trino has the best figures in 20 out of the 22 queries; the exceptions are queries number 11 (Drill is faster) and 17 (Spark is faster). The table also includes the average values of all the columns, so we can globally observe the relative differences among the systems. Thus, Trino is the fastest system with an average speed-up of 3.05, followed by Drill with 2.21, Spark with 1.79 and Hive with 1.09 being the slowest system.

About the performance of Spark, it is worth noting that roughly 15 to 20 seconds of the Spark execution time is spent on the Spark session initialization, which imposes a mandatory overhead in each execution. The relative influence of this penalization depends on the data size and the total computing time. This issue is analyzed in next section.

4.4.2 Scalability analysis

Once we have analyzed the TPC-H queries computing times, our interest is to determine whether the observed behavior persists when we scale the data size down (10 GB scale factor) and up (300 GB scale factor).

To summarize all the results in a single chart, we present in Fig. 4 a swarm plot for the execution times for each query, system, and scale factor. The lower limit of the y-axis is always 0.0 and the upper limit is highest time.

If we focus on the 10 GB data and compare them against the 100 GB ones, we can see that, in most of queries, the distances between the points corresponding to both datasets with Drill, Spark, and Trino are similar, so their rank does not change (i.e., Trino is the fastest, Spark is the slowest, Drill is remains in the middle). This is not the case for Hive, as its times with the 10 GB data are lower that the ones of Spark in many queries.

Since Hive is the worst performing system at the data scale factor of 100 GB, it can be intuited that its performance may degrade as the data size grows. This fact is confirmed when we take into consideration the 300 GB scale factor data, as we observe that, excepting some exceptions (queries 1, 2, and 6), Hive is by far the worst performing system. To explain this behavior, we have to take into account Hive's architecture. Hive's execution engine used is Tez which, in contrast to other frequent engines like MapReduce, does not write to disk at every step of the execution. However, according to the MR3 documentation, it uses "hostPath volumes" at local directories to hold intermediate data to be shuffled between worker pods. The more information, the higher the impact on performance this write has in the execution times.

On the contrary, we find that the performance of Spark improves when the size of the data grows. Thus, we observe that Spark becomes the fastest system in queries 4, 5, 7, 9, and 19, and its computing times are similar to the ones of Drill and Trino in queries 6, 8, 10, 12, 14, 16, 17, 20, and 22. We highlighted in the last section that Spark is penalized by a startup time in the order of several seconds. The results shown in Fig. 4 suggest that that overhead is not linear, and its influence on the total computing time is compensated when a high volume of data is processed.

An interesting case is query number 1, which presents significant time reductions for 100 and 300 GB scale factors. From Table 2 we observe that it is the query with the highest number of aggregation operations, three averages, one count, and four sums with group and order by clauses. An in-depth analysis of the benchmark from [19] suggests that aggregation implementations that use hash-tables to store group-by keys tend to spill to memory once these tables exceed the CPU cache levels. The optimizer should adopt a different approach that smoothes the cost to prevent that costly event. As the aggregated keys are stored in cache memory, subsequent queries experience a significant reduction in time.

Table 3 Mean execution time and speedup with respect to the slowest system for each of the TPC-H queries (100 GB scale factor)

Query	Drill		Hive		Spark		Trino	
	Mean	Speed-up	Mean	Speed-up	Mean	Speed-up	Mean	Speed-up
Q1	55.79	1.45	80.88	1.0	60.6	1.33	**47.16**	**1.71**
Q2			25.31	1.31	33.11	1.0	**16.85**	**1.97**
Q3	59.64	1.89	112.87	1.0	76.51	1.48	**55.11**	**2.05**
Q4	46.86	2.16	101.13	1.0	71.13	1.42	**31.0**	**3.26**
Q5	54.42	1.42	77.42	1.0	74.7	1.04	**54.34**	**1.42**
Q6	34.9	1.1	32.63	1.17	38.22	1.0	**23.47**	**1.63**
Q7	70.51	1.31	92.2	1.0	77.84	1.18	**50.6**	**1.82**
Q8	74.89	1.06	73.72	1.08	79.55	1.0	**52.11**	**1.53**
Q9	74.57	1.58	117.56	1.0	88.16	1.33	**54.66**	**2.15**
Q10	51.14	1.81	92.47	1.0	77.17	1.2	**36.47**	**2.54**
Q11	**11.45**	**2.51**	26.62	1.08	28.72	1.0	13.46	2.13
Q12	43.93	2.08	91.2	1.0	76.48	1.19	**30.44**	**3.0**
Q13	19.9	2.73	54.22	1.0	33.35	1.63	**11.09**	**4.89**
Q14	36.7	1.66	61.03	1.0	40.09	1.52	**27.04**	**2.26**
Q15	66.96	2.49	166.72	1.0	63.16	2.64	**46.13**	**3.61**
Q16	19.08	1.4	26.8	1.0	18.83	1.42	**7.5**	**3.58**
Q17	71.47	5.53	395.05	1.0	**63.47**	**6.22**	79.92	4.94
Q18	91.51	2.66	218.97	1.11	243.03	1.0	**62.65**	**3.88**
Q19	79.45	1.0	34.72	2.29	40.71	1.95	**24.48**	**3.25**
Q20	46.19	4.24	195.77	1.0	49.79	3.93	**35.93**	**5.45**
Q21	276.87	1.41	389.38	1.0	189.26	2.06	**123.93**	**3.14**
Q22	13.29	4.97	66.09	1.0	22.81	2.9	**9.73**	**6.79**
AVG.	61.88	2.21	115.13	1.09	70.3	1.79	**40.64**	**3.05**

The best results are highlighted in bold

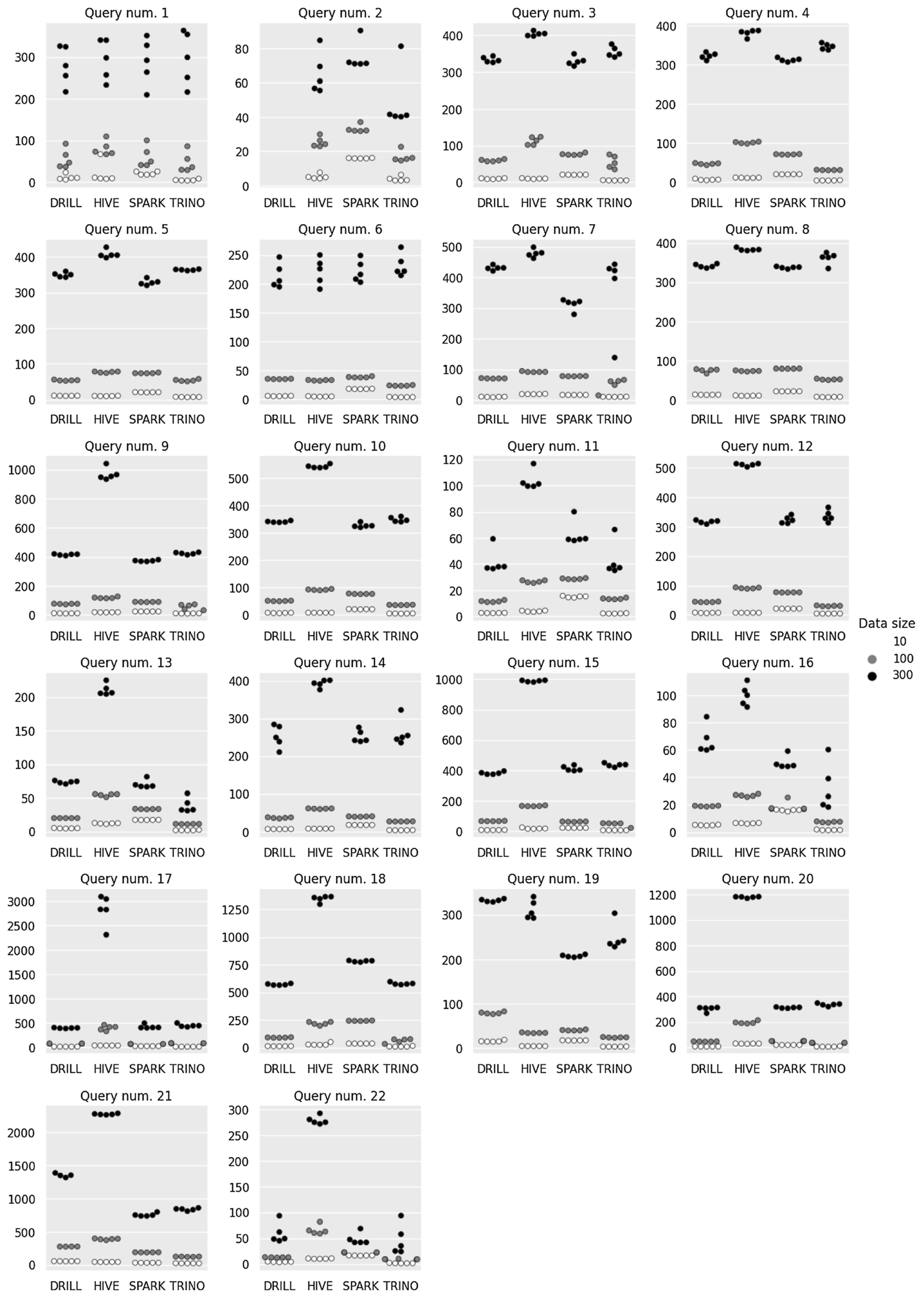


Fig. 4 Swarmplot of the computing times obtained when performing the TPC-H queries using different scale factors

4.4.3 Discussion

The four systems present a similar architecture for executing SQL queries. They consist of a master process and the several distributed workers, except Drill, which connects and syncs their workers through Zookeeper. Nevertheless, apart from their own local optimizations, policies or strategies, they handle data reading in a different fashion. Trino and Hive make use of the Hive Metastore for creating the SQL tables, having this as an external dependency, to which they connect through the JDBC protocol using Beeline.

As mentioned, Hive offers the worst performance as the data grows in size since it performs I/O for shuffling when executing queries. This behaviour is present across all queries except for queries 1, 2 and 6, where the results show that it provides a similar performance to its competitors:

- Q1 queries only table 'lineitem', which from Fig. 2, it can observed that is the table with the highest number of rows and columns (6.000.000 rows and 16 columns). This is reflected in the results from Fig. 3 and Table 3, from where it can be seen that despite accessing to a single table, it still is one of the queries that take a significant execution time from the benchmark.
- Q2 queries tables 'part', 'partsupp', 'nation', 'supplier' and 'region'. In spite of being several tables, they are relatively small in size compared to 'lineitem' or 'orders'. The table that has the highest number of rows is 'partsupp' (800.000) but it does have the low number of 5 columns.
- Q6, like Q1, only queries table 'lineitem'. However, it is computationally less costly since it does not perform any aggregations other than a sum and has no group by or nested queries that would increase its complexity.

From these observations we can extract that Hive works well when there are not join operations or nested queries with large data sizes. The only two queries that work with a single table are queries 1 and 6, where, despite having to query a large volume of data, it performs as well as its competitors. In the case of query 2, where there are nested queries, it is still able to perform well since most of the tables are relatively small in size and do not create a large overhead when writing the data into the hostPaths. In contrast to the other systems, which do have the spill-to-disk disabled, Hive's degradation in performance appears when the queries involve large tables with nested queries since they result in a high I/O bottleneck.

Trino consistently outperforms (for SF of 100 GB) the rest of the systems for almost every query as seen in Table 3. In comparison to its competitors, it does not seem to falter in any regard across the different aspects considered in the benchmark, except for a single execution in Q7 that crashed due to memory issues for SF of 100 GB (seen in Fig. 3), proving to be fast and reliable in most cases regardless of the type of query that is executed.

Following a different approach, Spark directly reads the data from HDFS through the creation of views. It lazily reads the data as required when executing queries. Drill can infer the schema from the data when reading from HDFS and also supports views. The latter approach was chosen since a view had to be created for some queries anyways.

This resemblance in the way of handling the data can be observed in queries 7, 10, 12, 13, etc. where, if we do not consider the constant initialization time that penalizes Spark, the execution times are significantly close. This suggests that the performance for both systems in most scenarios is similar. Nonetheless, it can be observed how Drill struggles against Spark with SC of 300 GB in some queries, e.g.: in Q21, which queries table 'lineitem' thrice, being the query with the largest data size from the benchmark. Regarding robustness, only Drill was not able to execute query number 2, independently of the scale factor.

5 Conclusions

This paper has compared four scalable SQL-on-Hadoop systems combined with Kubernetes: Apache Drill, Apache Hive, Apache Spark, and Trino. We have conducted not only a performance study but we have also included and discussed the deployment of the systems on Kubernetes, which was far from trivial due to the lack of official documentation and rather reduced communities around them.

The experimentation is based on the TPC-H benchmark queries, from which we have generated three datasets with different scaling factors: 10 GB, 100 GB and 300 GB. Our approach has been to analyze in detail the performance of the systems on the 100 GB dataset and then proceed with a scalability study including all three datasets. The TPC-H benchmark showed that Trino is in general the fastest system when working with the 10 GB and 100 GB data sets. Apache Spark has a initialization overhead that affects it negatively when the data size is not large enough, but when this is the case (in our study, with the 300 GB data set) it is the best performing system on many queries. Hive's computation times with smaller data sizes are comparable to its peers, but Hive suffers as data sizes increase, making it the worst scaling system by far. Drill appears as stable system, in the sense that it usually is not the best one but it is near the bests in most of the queries.

There are some lines for further research that emerge after the study carried out. The first one is to conduct an in-depth experimentation about the auto-scaling of the four

SQL systems on Kubernetes to determine how it behaves in certain scenarios, by setting a minimum and maximum number of nodes and analyzing how the nodes are added on the fly when the workload increases. We intend also to compare the performance the SQL systems on Kubernetes against a deployment of them on bare metal.

Author Contributions Conceptualization, CC, JFA, AMB, AJN, JMM, JJS.; Methodology, CC, JFA, AMB, AJN, JMM, JJS.; Software, CC, JFA, AMB.; Validation,CC,JFA, AMB, AJN, JMM, JJS.; Analysis, CC, JFA, AMB, AJN, JMM, JJS; writing-original draft preparation, CC, JFA, AMB, AJN.; writing-review and editing, CC, JFA, AMB, AJN All authors have read and agreed to the published version of the manuscript.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. Funding for open access charge: Universidad de Málaga / CBUA. This work has been partially funded by the Spanish Ministry of Science and Innovation via Grant PID2020-112540RB-C41 (AEI/FEDER, UE), Andalusian PAIDI program with grant P18-RT-2799, and by project "Evolución y desarrollo de la plataforma DOP de Big Data" (702C2000044) under Andalusian "Programa de Apoyo a la I+D+i Empresarial".

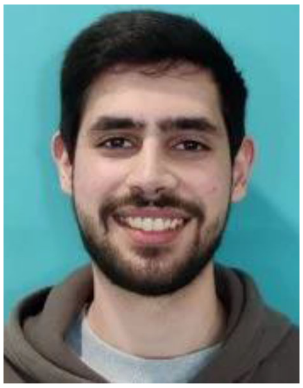
Data Availability The data and scripts are available at: https://doi.org/10.5281/zenodo.7059996.

Declarations

Conflict of interest The authors declare no competing interests.

References

1. White, T.: Hadoop: The Definitive Guide. O'Reilly Media, Sebastopol (2009)
2. Capriolo, E., Wampler, D., Rutherglen, J.: Programming Hive: Data Warehouse and Query Language for Hadoop. O'Reilly Media, Sebastopol (2012)
3. Russell, J.: Getting Started with Impala: Interactive SQL for Apache Hadoop. O'Reilly Media, Sebastopol (2014)
4. Fuller, M., Traverso, M., Moser, M.: Trino: The Definitive Guide. O'Reilly Media, Sebastopol, (2021)
5. Givre, C., Rogers, P.: Learning Apache Drill: Query and Analyze Distributed Data Sources with SQL. O'Reilly Media, Sebastopol (2018)
6. Chambers, B., Zaharia, M.: Spark: The Definitive Guide: Big Data Processing Made Simple. O'Reilly Media, Sebastopol (2018)
7. Abdollahi Vayghan, L., Saied, M.A., Toeroe, M., Khendek, F.: Deploying microservice based applications with kubernetes: Experiments and lessons learned. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 970–973 (2018)
8. Floratou, A., Minhas, U.F., Özcan, F.: Sql-on-hadoop: full circle back to shared-nothing database architectures. Proc. VLDB Endow. **7**(12), 1295–1306 (2014)
9. Poess, M., Floyd, C.: New TPC benchmarks for decision support and web commerce. SIGMOD Rec. **29**(4), 64–71 (2000)
10. Poess, M., Rabl, T., Jacobsen, H.-A.: Analysis of tpc-ds: The first standard benchmark for SQL-based big data systems. In: Proceedings of the 2017 Symposium on Cloud Computing. SoCC '17, pp. 573–585. Association for Computing Machinery, New York (2017)
11. Chen, Y., Qin, X., Bian, H., Chen, J., Dong, Z., Du, X., Gao, Y., Liu, D., Lu, J., Zhang, H.: A study of SQL-on-Hadoop systems. In: Zhan, J., Han, R., Weng, C. (eds.) Big Data Benchmarks, Performance Optimization, and Emerging Hardware, pp. 154–166. Springer, Cham (2014)
12. Tapdiya, A., Fabbri, D.: A comparative analysis of state-of-the-art sql-on-hadoop systems for interactive analytics. In: Nie, J., Obradovic, Z., Suzumura, T., Ghosh, R., Nambiar, R., Wang, C., Zang, H., Baeza-Yates, R., Hu, X., Kepner, J., Cuzzocrea, A., Tang, J., Toyoda, M. (eds.) 2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11–14, 2017, pp. 1349–1356. IEEE Computer Society, Boston (2017)
13. Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S., Stonebraker, M.: A comparison of approaches to large-scale data analysis. SIGMOD '09, pp. 165–178. Association for Computing Machinery, New York (2009)
14. Rodrigues, M., Santos, M.Y., Bernardino, J.: Big data processing tools: an experimental performance evaluation. WIREs Data Min. Knowl. Discov. **9**(2), 1297 (2019)
15. Aluko, V., Sakr, S.: Big sql systems: an experimental evaluation. Clust. Comput. **22**, 1347–1377 (2019)
16. Zhu, C., Han, B., Zhao, Y.: A comparative study of spark on the bare metal and kubernetes. In: 2020 6th International conference on big data and information analytics (BigDIA), pp. 117–124 (2020)
17. Raju, A., Ramanathan, R., Hemavathy, R.: A comparative study of spark schedulers' performance. In: 2019 4th international conference on computational systems and information technology for sustainable solution (CSITSS), vol. 4, pp. 1–5 (2019)
18. Transaction Processing Performance Council TPC: TPC Benchmark H (Decision Support) Standard Specification (2021)
19. Boncz, P., Neumann, T., Erling, O.: TPC-H analyzed: hidden messages and lessons learned from an influential benchmark. In: Nambiar, R., Poess, M. (eds.) Performance Characterization and Benchmarking, pp. 61–76. Springer, Cham (2014)



Cristian Cardas is pursuing Ph.D. in the Department of Languages and Computer Sciences at the University of Malaga (UMA), Spain. He received his Bachelor of Computer Science Engineering in 2020 and Master in Software Engineering and Artificial Intelligence in 2021 at UMA. He is part of ITIS (Instituto de Tecnología e Ingeniería del Software) and Khaos Research Group, where he conducts his research. His current research interests are Precision Agriculture, Artificial Intelligence and Machine Learning.

José F. Aldana-Martín has received his BSc in Software Engineering in 2020 and his MSc in Software Engineering and Artificial Intelligence in 2021, both at the University of Málaga, Spain. Currently, he is pursuing a PhD in the Department of Languages and Computer Science at the University of Málaga, conducting his research as part of Khaos Research and ITIS (Instituto de Tecnología e Ingeniería del Software). His research interest includes Multi-objective Optimization, Semantic Web, Remote Sensing and Artificial Intelligence.

Antonio M. Burgueño-Romero was awarded a BS in Computer Science in 2020, followed by his MS in Software Engineering and Artificial Intelligence in 2021 (both from the University of Málaga, in Spain). Burgueño has papers published in a wide variety of fields, such as: Remote Sensing, Computer Vision, Robotics or Artificial Intelligence. Right now, he is focused in the areas of MLOps and distributed applications.

Antonio J. Nebro received the M.S. and Ph.D. degrees in Computer Science from the University of Málaga, Málaga, Spain, in 1992 and 1999, respectively. He is currently a Full Professor of Computer Science with the Departamento de Lenguajes y Ciencias de la Computación, University of Málaga, and member of ITIS (Instituto de Tecnología e Ingeniería del Software) and Khaos Research Group. He has published more than 40 scientific papers in international indexed journals, more than 55 papers in international conferences, and co-authored 17 book chapters. His current research interests include multi-objective optimization, optimization frameworks design and implementation, parallel metaheuristics, and optimization applied to real-world problems.

Jose M. Mateos is working as Software Development Leader at Ericsson where, among other responsibilities, he has being managing a SW Architecture team, whose main focus was the development of network design and optimization software. With his team, he contributed to the design and development of big data platforms for processing and analyzing radio access network information. During the last ten years, he has actively being contributing to the development of SW tools for network design and optimization.

Juan J. Sánchez is working as Software Architect at Ericsson focusing on network design and optimization software participating in the design and development of big data platforms for processing and analyzing radio access network information. Previously he has worked as a researcher in parallel algorithms for high-performance computing and massive genomic data processing over clusters with several presentations at the Bioinformatics congress. He obtained a Computer Engineering degree and a Master's in Software Engineering and Artificial Intelligence from the University of Málaga.