Investigating Automatic Parameter Tuning for SQL-on-Hadoop Systems

Edson Ramiro Lucas Filho[a,⋆], Eduardo Cunha de Almeida[b], Stefanie Scherzinger[a] and Herodotos Herodotou[c]

[a] *University of Passau, Germany*
[b] *Federal University of Paraná, Brazil*
[c] *Cyprus University of Technology*

ARTICLE INFO

Keywords:
SQL-on-Hadoop
Parameter Tuning
Self-Tuning

ABSTRACT

SQL-on-Hadoop engines such as Hive provide a declarative interface for processing large-scale data over computing frameworks such as Hadoop. The underlying frameworks contain a large number of configuration parameters that can significantly impact performance, but which are hard to tune. The problem of automatic parameter tuning has become a lively research area and several sophisticated tuning advisors have been proposed for Hadoop. In this paper, we conduct an experimental study to explore the impact of Hadoop parameter tuning on Hive. We reveal that the performance of Hive queries does not necessarily improve when using Hadoop-focused tuning advisors out-of-the-box, at least when following the current approach of applying the same tuning setup uniformly for evaluating the entire query. After extending the Hive query processing engine, we propose an alternative tuning approach and experimentally show how current Hadoop tuning advisors can now provide good and robust performance for Hive queries, as well as improved cluster resource utilization. We share our observations with the community and hope to create an awareness for this problem as well as to initiate new research on automatic parameter tuning for SQL-on-Hadoop systems.

1. Introduction

The increasing need to process analytical queries over large-scale semi-structured data has led to the development of SQL-on-Hadoop engines [17, 9]. These systems evaluate SQL-like queries over data stored in distributed file systems such as the Hadoop Distributed File System (HDFS) [47]. Hive [53] was the first SQL-on-Hadoop system to provide an SQL-like query language, namely HiveQL, and can use MapReduce or Tez as its underlying framework for executing queries. Shark [59] and Spark SQL [2] also support HiveQL but use the Spark framework as their runtime instead of MapReduce. On the other hand, Impala [29] uses its own processing framework to execute queries, bypassing the MapReduce computing model in order to provide better support for interactive queries. Similar to Impala, the big data engines Apache Tajo [50], Presto [43], and Drill [14] also use a custom runtime to execute SQL queries following a shared-nothing parallel database architecture.

The performance of the underlying computing frameworks (such as MapReduce and Spark) and custom runtimes (e.g., of Impala, Tajo) is governed via a large number of configuration parameters that control memory distribution, I/O optimization, task parallelism, and data compression [22, 13, 5, 21]. Both MapReduce and Spark have over 200 parameters each, out of which 20–30 can have significant impact on the performance and stability of the cluster. Several studies have shown that jobs can experience up to an order of magnitude difference in execution time between good and bad parameter settings [3, 26]. However, regular users and even expert administrators struggle to understand and tune them to achieve good performance. A recent report highlights that the proliferation of MapReduce goes hand-in-hand with continuous lamentations regarding the lack of professionals who can tune a Hadoop cluster [23]. This skills gap has given rise to a successful line of research on automatically tuning MapReduce and Spark parameters using a variety of techniques such as cost models, simulation, and machine learning [37, 35, 46, 38, 6, 34, 5, 55, 22, 15], including techniques for tuning SQL-on-Hadoop systems and virtual machines together [39].

In this paper, we conduct an experimental study focused on Hive over Hadoop MapReduce because (i) Hive is a good representative of native SQL-on-Hadoop systems; (ii) both Hive and Hadoop are highly popular for analytical processing; and (iii) Hadoop parameter tuning has been studied extensively in recent years [17, 9, 37, 6]. Our goals are to (i) explore the impact of Hadoop parameter tuning on Hive, (ii) identify the potential use of existing Hadoop tuning advisors for optimizing Hive performance, and (iii) propose promis-

⋆Corresponding author
edson.lucas@uni-passau.de (E.R.L. Filho); eduardo@inf.ufpr.br (E.C.d. Almeida); stefanie.scherzinger@uni-passau.de (S. Scherzinger); herodotos.herodotou@cut.ac.cy (H. Herodotou)
ORCID(s): 0000-0002-0536-5506 (E.R.L. Filho); 0000-0002-6644-956X (E.C.d. Almeida); 0000-0002-8717-1691 (H. Herodotou)

ing directions in parameter tuning for SQL-on-Hadoop systems. For tuning Hadoop, we used Starfish [22, 19], the first cost-based optimizer for finding (near-) optimal configuration parameter settings and the only publicly available tuning advisor for academic research purposes. In addition, Starfish achieves higher reported speedups compared to other tuning advisors (see Table 4, Appendix A). Finally, since we did not modify the Starfish Optimizer when implementing our approach, it would be straightforward to replace Starfish with another Hadoop tuning advisor.

Our results show that Hadoop parameter tuning can have a drastic impact on both the performance of HiveQL queries and the efficient utilization of cluster resources. Given that Hive compiles HiveQL queries into a workflow of MapReduce jobs, it would be straightforward to assume that by tuning the underlying Hadoop framework, HiveQL queries would benefit as well. However, we show that this assumption does not hold when using an existing Hadoop tuning advisor naively, due to the architecture and design choices of Hive, Hadoop, and the tuning advisors as well. This in turn has lead us to extend Hive for automatically tuning each generated MapReduce job individually. Even though this study focuses on Hive, we believe that the presented problem generalizes to other SQL-on-Hadoop systems as well (discussed in Section 7).

To the best of our knowledge, this is the first study on parameter tuning for SQL-on-Hadoop systems. Two previous studies [17, 9] compared the execution time of TPC-H and TPC-DS like queries across different SQL-on-Hadoop offerings, namely Hive, Impala, Stringer, Presto, and Shark. However, none of them considered parameter tuning nor investigated cluster resource utilization patterns.

Our core contributions in this paper are as follows:

1. We study the impact of parameter tuning on SQL-on-Hadoop systems (namely Hive) and show that the approach taken by current Hadoop tuning advisors is not directly applicable for tuning Hive.
2. We explore an alternative approach and experimentally show how a current Hadoop tuning advisor can be made to provide good and robust performance for HiveQL queries.
3. We investigate the impact of parameter tuning to cluster resource utilization and observe that our proposed tuning strategy can significantly reduce the resource usage of HiveQL queries.
4. We present open research problems and a research agenda towards automatic parameter tuning for SQL-on-Hadoop systems.

The rest of the paper is organized as follows: Section 2 reviews query execution in Hive and Hadoop, with a focus on tuning. Section 3 presents our execution environment. Sections 4, 5, and 6 present and analyze the experimental results, while Section 7 outlines key lessons learned and discusses promising directions for future work.

2. Preliminaries

We review the preliminaries of Hive query execution and the current state-of-the-art in Hadoop parameter tuning.

2.1. Query Execution in Hive

Hive [53], an open source project originally developed at Facebook, was the first SQL-on-Hadoop query engine built on top of Hadoop MapReduce to leverage its scalability, fault tolerance, and scheduling features. Hive also introduced HiveQL, a SQL-like query language that has become the standard SQL interface for large-scale data [17].

Hive internally uses a query optimizer that resembles traditional relational optimizers to translate a given HiveQL statement to a workflow of MapReduce jobs, to be executed on Hadoop. Specifically, a HiveQL statement is first parsed and validated against the data dictionary, compiled into a tree of logical operators, and finally optimized to produce a physical query execution plan [17, 53]. This allows for various logical optimizations, such as selection and projection pushdown, as well as join reordering and physical join operator selection. Physical tuning, such as partition pruning, is also performed when an input table is split over several HDFS folders, to avoid processing unnecessary data [24]. The final execution plan has the form of a Directed Acyclic Graph (DAG) of MapReduce *jobs*. Each job is executed on the cluster as a set of parallel Map and Reduce *tasks*.

As a further, MapReduce-specific optimization, the query optimizer merges jobs by applying *chain folding* and *job merging*, which are both considered generic MapReduce design patterns [40]. As a simple example, chain folding collapses sequences of Map-only jobs into a single Map-only job[1]. Job merging further merges jobs that process the same input data, such as joins applying a common join predicate. Hence, *each resulting MapReduce job will typically evaluate several query operators*. Similar optimizations have also been proposed in Stubby [36], YSmart [32], and MRShare [41], as they are very effective in speeding up query execution by reducing the number of intermediate files that are written out to HDFS in between MapReduce jobs. In MapReduce processing, such communication costs can dominate execution times [44].

Figure 1 presents the query execution plan for TPC-H query 7 consisting of a DAG of six MapReduce jobs. This particular DAG has the form of a tree, where jobs 1 and 2 are the leaves and the final job 6 is the root. As can be seen, each MapReduce job evaluates several query operators (e.g., Job 2 contains two TableScans

[1] In Hive and Spark terminology, merged jobs are *stages*. We follow the convention of Hadoop processing and use the general term *job*.

Table 1
Selected parameter values generated by Starfish for each job of TPC-H query 7 and the default (out-of-the-box) tuning values shown on the right. (*Job-1 is a map-only job; hence, `mapred.reduce.tasks` is not set. †By default, Hive overrides `mapred.reduce.task` based on the input size of each job.)

Tuning Knob	Description	Job-1	Job-2	Job-3	Job-4	Job-5	Job-6	Def.
io.sort.mb	Buffer size for sorting map output	100	879	593	593	1050	547	100
io.sort.factor	Number of streams to merge during map-side sorting	10	32	64	64	71	22	10
io.sort.record.percent	Fraction of io.sort.mb for storing metadata	0.05	0.02	0.01	0.01	0.40	0.23	0.05
io.sort.spill.percent	Threshold usage of io.sort.mb for beginning sort	0.80	0.60	0.78	0.78	0.64	0.51	0.80
mapred.inmem.merge.threshold	Map-side threshold for merging data during shuffle	1000	470	651	651	698	568	1000
mapred.job.reduce.input.buffer.percent	% of memory used to buffer map output during reduce	0.00	0.17	0.30	0.30	0.44	0.24	0.00
mapred.job.shuffle.input.buffer.percent	% of memory used to buffer map output during shuffle	0.70	0.30	0.38	0.38	0.40	0.51	0.70
mapred.job.shuffle.merge.percent	Reduce-side threshold for merging data during shuffle	0.66	0.68	0.82	0.82	0.48	0.51	0.66
mapred.reduce.tasks	Number of reduce tasks	–*	36	36	36	8	8	1†

and a Join, among others). The output of each job is written to a temporary HDFS file, to be processed by the next upstream job in the workflow.

2.2. Tuning Advisors for Hadoop

The execution behavior of MapReduce jobs in Hadoop is governed by over 200 configuration parameters, out of which 20 can have a significant impact on performance [22, 13, 5, 21]. Table 1 lists some of the most important Hadoop parameters that control memory settings, thread scheduling, and task parallelism. A notable amount of work has been done over the last decade on automating the selection process, i.e., *tuning* of Hadoop parameters. This lead to the creation of several *tuning advisors*, as listed in Table 4 (see Appendix A). Note that this work does not aim at benchmarking or comparing tuning advisors, but rather, at analyzing how to best apply them for tuning SQL-on-Hadoop systems.

The high level goal of a Hadoop tuning advisor is to propose a set of parameter values that minimizes execution time and/or improve resource utilization of the cluster. For brevity, we will refer to a set of specific values for Hadoop parameters as a *tuning setup*. Tuning advisors consider the data flow, processing times, and access costs to resources, e.g., tracking job counters [35, 34], real-time statistics [13], job execution times [5] or the execution times of the MapReduce phases [6], instrumentation of the JVM [37, 22], and execution logs [46, 51] for proposing tuning setups. We will collectively call such information the *execution profile* of a MapReduce job. For this study, we utilize the execution profiles collected by Starfish, described in [19].

Given an execution profile, the tuning advisor employs a modeling technique (e.g., cost/analytical modeling, machine learning) to estimate the execution time of the given job under a given tuning setup [18, 63, 49, 20, 10]. Next, the tuning advisor will enumerate and search over the high-dimensional space of parameter values in order to identify the tuning setup that will produce the smallest execution time. Different advisors will employ different search strategies, ranging from grid search and exhaustive enumeration to recursive random search, hill climbing, and genetic algorithms

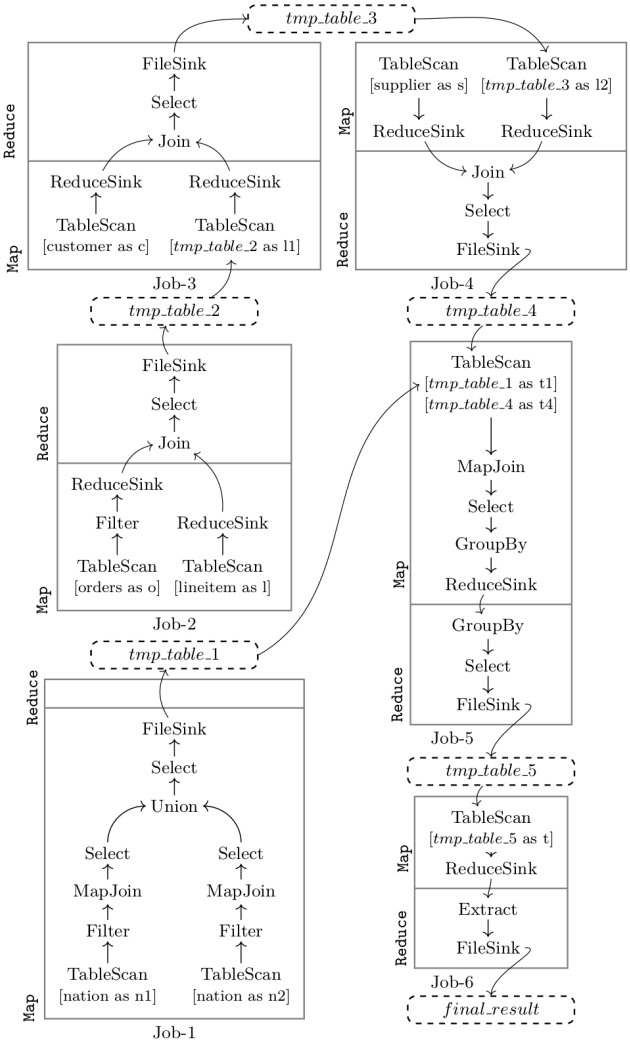


Figure 1: Hive query plan for TPC-H query 7.

(see Table 4).

The workloads chosen for evaluating Hadoop tuning advisors typically consist of simple jobs such as *Sort*, *Word Count*, and *Grep*, whereas very few systems report the use of handcrafted HiveQL queries or employ a small subset of TPC-H and TPC-DS (listed in Table 4, Appendix A). Hence, most experiments evaluate advi-

sors for MapReduce jobs that implement or mimic a single SQL operator. Such microbenchmarking means that only a single scenario is examined, whereas the workloads generated by the Hive query optimizer implement several query operators per MapReduce job. Consequently, it is not clear how well tuning advisors perform against complex SQL-like analytical workloads. In this paper, we explore a *macrobenchmark* examination of query execution that includes the evaluation of query plans with many HiveQL operators.

2.3. Parameter Tuning in Hive

For tuning, a Hive administrator may configure up to 200 parameters for the underlying Hadoop system. The authors in [42, 45] show how sensitive these settings are to the software and hardware stack. Even different implementations of the JVM (e.g., OpenJVM vs. IBM JVM) impact performance [11]. Yet, the problem of automatically setting these parameters for Hive (or other SQL-on-Hadoop systems) remains largely unexplored today.

A Hive administrator can assign a tuning setup to a query plan in the query code, or when submitting the query via the command line. The Hive query processing engine then propagates this tuning setup to all MapReduce jobs in the query plan, i.e., at the level of individual HiveQL queries, *all* MapReduce jobs that constitute a query plan are executed with *identical* Hadoop configuration settings. Hence, when administrators are tasked with tuning a query, they must find a single setting of Hadoop parameter values to use on a per-query basis, or even for the complete query workload. Even with the help of a Hadoop tuning advisor, the same tuning setup will still be replicated to the entire query plan. The overall workflow of uniform tuning is shown in Figure 5(a). Out-of-the-box, Hive does not support a more fine-grained tuning (e.g., on a per-job basis). We refer to this approach as *uniform tuning* and will discuss it in more detail in Section 4.

3. Experimental Setup

Our execution environment is a cluster of ten (10) m5.2xlarge machines[2] hosted at Amazon Web Services (AWS), each one with 8 CPU cores, 32 GiB of RAM, and 500 GiB of dedicated EBS disk with up to 4750 Mbps bandwidth. The cluster runs Ubuntu 16.04 with Hive 0.13.1, Hadoop 0.22.2, and Starfish 0.3[3]. These specific Hadoop and Hive versions are those supported by the latest version of Starfish. The HDFS replication factor is set to 3 and the default block size is 128MB. The maximum number of Map and Reduce tasks that can be started on each node is set to fit the maximum capacity of the CPU (8 threads per node).

For our evaluation, we use TPC-H [54], a standard decision support benchmark consisting of 22 queries over 8 tables. Even though TPC-H was developed for evaluating traditional relational database systems, it is now widely used for evaluating SQL-on-Hadoop engines [17]. The TPC-H queries contain operations ranging from simple selection to complex joins and aggregations; thus, the generated MapReduce jobs exhibit a wide variety of characteristics. We generated the data with scale factor of 200, which yields 200GB.

Table 2
TPC-H queries with their HiveQL Operators (aggregated), number of MapReduce jobs, number of Map and Reduce tasks, input data size, and run time in our setup.

	HiveQL Operators										Runtime information				
Query	Extract	File Sink	Filter	Group By	Join	Limit	Map Join	Reduce Sink	Select	Table Scan	Jobs	Map Tasks	Reduce Tasks	Input (GB)	Time (sec)
1	1	2	1	2	0	0	0	2	3	2	2	608	162	149.44	231.0
2	1	6	1	2	1	1	4	4	4	7	6	153	38	27.71	123.7
3	1	4	3	2	2	1	0	6	2	6	4	772	205	187.35	349.3
4	1	4	2	4	1	0	0	5	5	5	4	758	202	182.76	323.7
5	1	7	1	2	2	0	3	6	6	9	7	822	56	187.62	348.0
6	0	1	1	2	0	0	0	1	2	1	1	601	1	149.44	125.3
7	1	6	3	2	3	0	3	8	8	10	6	972	259	187.62	620.0
8	1	8	2	2	2	0	5	6	8	10	8	817	209	192.20	503.5
9	1	7	1	2	3	0	2	8	6	10	7	1473	390	210.47	1321.0
10	1	5	2	2	2	1	1	6	2	7	5	794	207	187.35	347.5
11	1	5	1	4	0	0	3	3	5	5	5	108	3	23.13	90.7
12	1	3	1	2	1	0	0	4	3	4	3	745	199	182.76	240.7
13	1	4	1	4	1	0	0	5	4	5	4	164	44	37.91	185.0
14	0	2	1	2	1	0	0	3	2	3	2	629	167	154.02	196.3
15	1	3	1	4	0	0	2	3	5	3	4	618	164	149.70	206.7
16	1	5	3	2	1	0	1	4	5	6	6	181	38	27.71	141.3
17	0	4	2	4	1	0	1	4	5	5	4	1240	330	154.02	678.8
18	1	5	1	4	2	1	0	8	4	8	5	1463	389	187.35	709.7
19	0	2	1	2	1	0	0	3	2	3	2	630	167	154.02	408.0
20	1	6	3	6	0	0	4	4	8	6	6	743	171	177.15	370.0
21	1	9	5	6	3	1	2	10	10	12	9	2015	376	183.02	964.5
22	1	7	4	6	0	0	2	4	9	7	7	169	40	37.91	204.3

We used the TPC-H benchmark queries in the version issued for Hive[4]. Some TCP-H queries consist of several HiveQL statements (e.g., Q15). In this case, we report the aggregated values. Table 2 summarizes the information for the 22 TPC-H queries executed with the default Hadoop configuration, listing the number of relevant HiveQL operators executed, the number of MapReduce jobs generated[5], the number of Map and Reduce tasks, the input data size, and the overall run time (i.e., the aggregated run times taken by Hadoop to execute the jobs of the query plan). As required by Starfish, all queries were profiled once using the default Hadoop configuration. All reported run times and statistics are averaged over 3 runs. In the remainder of the paper, all speedups are reported as the percentage of the run times w.r.t. the default configuration (our baseline), shown in the last column of Table 1.

[2] https://aws.amazon.com/ec2/instance-types/m5/
[3] https://www.cs.duke.edu/starfish/release.html
[4] https://issues.apache.org/jira/browse/HIVE-600
[5] Hive can evaluate parts of a query using local jobs rather than MapReduce jobs. We ignore such local jobs in our discussion.

4. Uniform Parameter Tuning

In this section, we present the impact of *uniform tuning* on the execution of queries. In uniform tuning, all MapReduce jobs for evaluating a HiveQL query are run with the same, identical tuning setup (except for setting the number of reduce tasks, as we will discuss shortly).

4.1. Experimental Methodology

The goal of uniform tuning is to find a good tuning setup to use for all MapReduce jobs that comprise a given HiveQL query. However, current Hadoop tuning advisors, such as Starfish, only propose tuning setups for individual MapReduce jobs. Hence, an intuitive strategy is to use Starfish to generate one good tuning setup for each job in the query plan, and then select one of those tuning setups for executing the entire query. The key rationale here is that the selected tuning setup will improve the performance of at least one MapReduce job and, hopefully, of the query as a whole.

For instance, let us consider query 7, with the query plan shown in Figure 1. Starfish produces six candidate tuning setups (shown in Table 1) based on the six MapReduce jobs of the query plan. Note that job-1 is a map-only job and, since the parameters listed in Table 1 apply only to jobs with both Map and Reduce phases, Starfish does not propose specific values for any of these parameters. We analyze the performance induced by all candidate tuning setups in Section 4.2, and discuss alternative strategies for selecting a tuning setup for a query in Section 4.3.

4.2. Performance of Candidate Tuning Setups

We begin our discussion by focusing on the impact of the six candidate tuning setups on query-7's performance, which ranges from a 35.2% slowdown to an 11% speedup in query execution time (as shown in Figure 2). To better understand the effect of the parameter values in the different tuning setups, let us examine the number of reduce tasks for query 7, one of the most impactful Hadoop parameters [1]. In the baseline configuration, Hadoop by default sets the number of reduce tasks to 1 (see Table 1). However, Hive overrides the Hadoop default configuration and sets the number of reduce tasks per job to the size of its input data, divided by 256MB.[6] The number of reduce tasks is the only parameter that Hive controls at the granularity of single jobs. In the execution of query 7, Hive does not set the number of reduce tasks for job-1, because it is a map-only job. Hive further sets 197, 25, 19, 17, and 1 reduce tasks for jobs 2–6, respectively. With this configuration, Hive executes query 7 in 620 seconds.

Next, let us consider the tuning setups generated by Starfish. Starfish recommends 36 reduce tasks for jobs 2–4, and 8 reduce tasks for jobs 5 and 6 (see Table 1). The number of reduce tasks recommended by Starfish diverges considerably from the estimation made by Hive, as Starfish uses more fine-grained information to calculate the tuning setups.

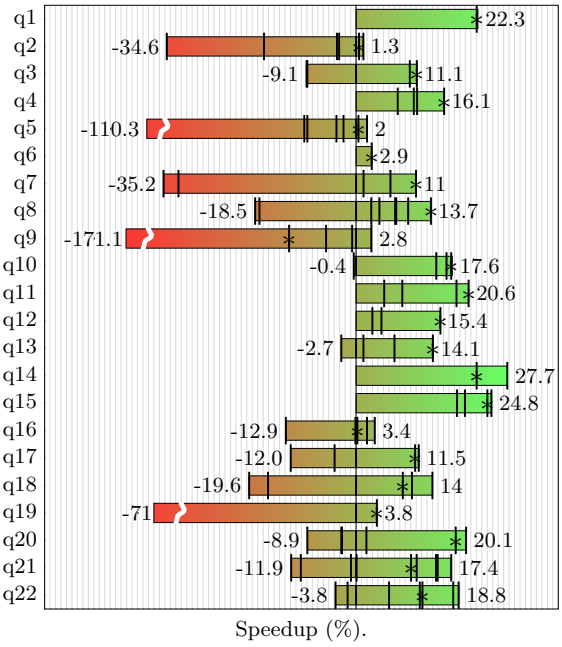


Figure 2: Speedups of TPC-H queries achieved by uniform tuning (in %) for each candidate tuning setup (listed in Tables 5 and 6 in Appendix A). The tuning setups generated for the dominant job (in terms of run time) are marked by *.

Following our experimental methodology, we apply each tuning setup generated by Starfish in turn. We will address the tuning setup generated for job-1 as tuning-1, for job-2 as tuning-2, and so on. When applying tuning-1 for executing the entire query, the number of reduce tasks is not set by Starfish because job-1 is a map-only job. Consequently, Hive estimates the same number of reduce tasks as in the default configuration (presented above). Tuning-1 achieves 1.37% of speedup, which is negligible and predictably close to the baseline. Tuning-2 achieves 11% of speedup and is the best case for query 7. Job-2 runs in 300 seconds using default settings, it is the longest running job in query 7, and accounts for 48.4% of the overall query run time. Interestingly, we have observed that using the tuning setup for the job with the longest running time is the best option for query 7. However, as common with heuristics, this strategy does not work for all queries, as we demonstrate in Section 4.3. Tuning-3 and tuning-4 achieve about 6% of speedup. Both jobs take about 125 seconds each and together make up around 40% of the overall run time. Tuning-2, 3, and 4 set the number of reducers to 36, but the resulting run times are different due to the fine-grained configuration of the other parameters. The performance of the queries does not rely on the number of reduce tasks alone, but

[6] https://github.com/apache/hive/blob/master/ql/src/java/org/apache/hadoop/hive/ql/exec/Utilities.java#L3090

on the tuning setup as a whole. Finally, both tuning-5 and tuning-6 set the number of reducers to 8, degrading performance by 32.5% and 35.24%, respectively.

We replicated this process for all 22 TPC-H queries, and present in Figure 2 the speedup of each tuning setup relative to the default configuration. Query 6 is the only query comprising a single job. All other queries have more than one job and, consequently, more than one tuning setup. In Figure 2, we represent the speedup of each tuning setup by a black vertical bar. The asterisk (¡) marks the speedup of the tuning setup generated for the job with the longest run time within a query, i.e., the *dominant job* (elaborated further in Section 4.3). As observed in Figure 2, the run times vary considerably with uniform tuning. Speedups range from 1.3% in query 2 to 27% in query query 14. Slowdowns range from -171.1% in query 9 to -0.4% in query 10. It is interesting to note that there are a total of 39 tuning setups (out of 107) that degrade performance, for 15 out of the 22 queries.

4.3. Strategies for Selecting a Tuning Setup

System administrators have several options for assigning tuning setups to Hive queries: (1) to rely on intuition (or chance) for selecting a good tuning setup; (2) to rely on heuristics, such as selecting the tuning setup of the longest running job in the query; or (3) to exhaustively test-run all candidate tuning setups.

(1) Feeling lucky: Although selecting a tuning setup at random is not a realistic strategy, it highlights that selecting the wrong tuning setup—even from the tuning setups generated for the query plan by an optimizer—can severely degrade performance. For instance, tuning-6 of query 7 degrades performance by 35.24%, while the worse-case scenario for query 9 is -171.1% (i.e., 2.7× slower). On the one hand, the probability of degrading performance is the number of bad tuning setups divided by the number of available tuning setups, which is 33% in the case of query 7. On the other hand, the probability of selecting the (near-) optimal tuning setup is 1 divided by the number of available tuning setups, which is only 16% in the case of query 7.

The 22 TPC-H queries produce 107 jobs in total with a 20% probability of selecting the (near-) optimal tuning setups for each query. Thus, relying on chance in selecting a tuning setup is likely to produce a tuning setup where query execution will under-perform, or even significantly degrade.

(2) Tuning the dominant job: One natural strategy is to select the tuning setup generated for the job dominating the runtime. The intuition behind this strategy is that the query runtime would improve by decreasing the runtime of the longest running job, even if the performance of some of the smaller jobs degrade. In Figure 2, the runtimes achieved with this strategy are marked with an asterisk (*). As this figure illustrates, the speedup achieved with this strategy can vary significantly as in some cases tuning the dominant job leads to the best case of uniform tuning, while in others it actually degrades performance. As a mid-case example, let us consider query 21, the query with the highest number of jobs (nine jobs in total[7]). The dominant job in query 21 is job-1, which consumes 32% of the total run time. The tuning setup generated for job-1 achieves only 10% of speedup. However, the tuning setup generated for job-2 performs better, achieving 18.8% of speedup, even though job-2 amounts to 24% of the total run time (and thus less than job-1).

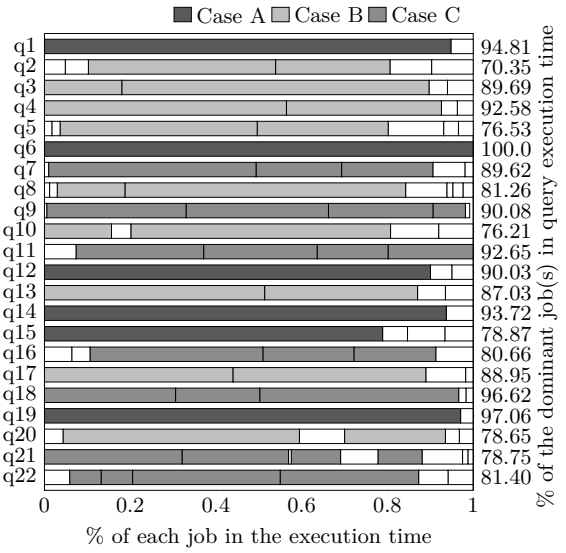


Figure 3: Execution breakdown of jobs in the query run time (run with default configuration). Right: Percentage of the dominant job(s) in the query run time. Jobs are classified into cases A–C, depending on their share of query run time.

To further study this strategy, we organize the TPC-H queries into 3 different cases, when the query: (A) has one job that dominates query execution time. (B) has 2 jobs, where the sum of their run times dominates the query execution time, and (C) has 3 or more jobs, where the sum of their run times dominates query execution time. For this experiment, we say that a job or jobs *dominate* query execution time when their total run time constitutes at least 70% of the total query execution time. Figure 3 visualizes the share of individual jobs in the query execution time and presents the percentage of the dominant job (or jobs) in the total query execution time. The cases A, B, and C contain 6, 9, and 7 queries, respectively. Thus, the cases are of roughly similar size.

Figure 4 summarizes the maximum, average, and minimum speedup achieved via uniform tuning (using the tuning setup of the dominant job) for the three cases. Observe that uniform tuning tends to be very effective in case A, because most of the queries in this

[7]Run times of each job of query 21 (seconds): job-1: 309, job-2: 240, job-3: 6, job-4: 111, job-5: 84, job-6: 99, job-7: 91, job-8: 12, job-9: 12, total of 964 seconds.

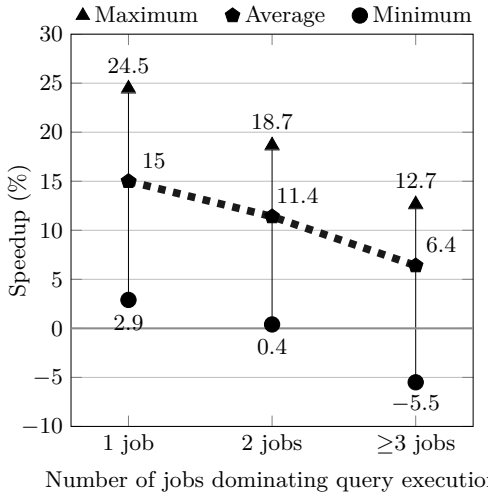


Figure 4: Maximum, average, and minimum speedup of uniform tuning (using Option 2) for queries with 1, 2, 3, or 4 dominant (*i.e.*, long running) jobs within the same query plan.

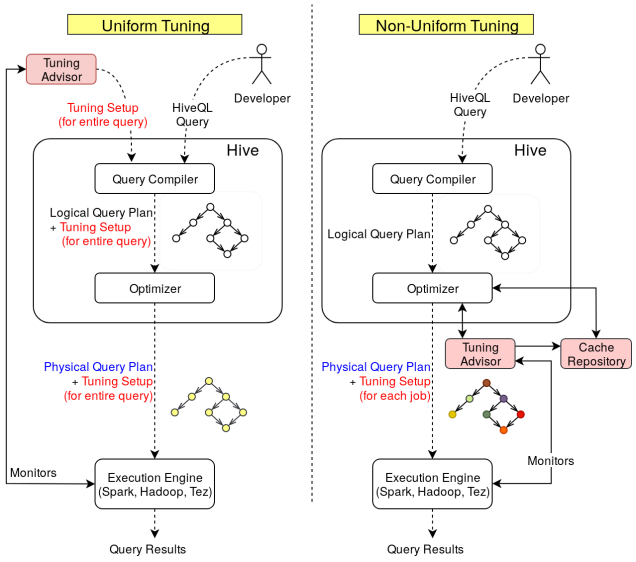


Figure 5: (a) Uniform and (b) Non-uniform tuning workflow.

case have one single job dominating even over 90% of query execution time. Queries in case A resemble single MapReduce jobs for which Starfish can generate good tuning setups. However, only 6 out of the 22 TPC-H queries fall into case A. The dashed trend line indicates that as the number of long-running jobs increases in a query, the strategy of uniform tuning based on the longest running job becomes less effective. Note that the average speedup decreases by about half from case B to C. In case C, tuning setups generated for the dominant jobs may even degrade performance (observed for queries 9 and 16). Hence, one tuning setup generated for a specific job is not necessarily effective for other jobs. It is a clear case of "one size does not fit all".

(3) Exhaustive test-run: The last option for selecting a tuning setup is to exhaustively test-run all candidate tuning setups, as we did in our experiments. We profile the queries (a first complete execution) for generating the tuning setups and then we test-run all setups. For instance, query 21 with 9 jobs requires 10 executions: one for profiling and generating the tuning setups, and 9 runs for testing. In total, the 22 TPC-H queries produce 107 jobs and require 129 test runs. The test-runs take 7.5× more time than running with the default configuration, which is rather impractical, as also confirmed in discussions with Hive practitioners. In a few specific scenarios, such as running an analytics workload repeatedly in a static environment, it may be acceptable to spend so much time in test-runs. However, dynamic aspects like the growth/change of data sets as well as updates in the hardware and software stacks are common in the life cycle of SQL-on-Hadoop systems and would require administrators to re-evaluate the test-runs frequently.

5. Non-uniform Parameter Tuning

In this section, we present *non-uniform* tuning as an alternative to *uniform* tuning: Each MapReduce job in a HiveQL query is executed with its own tuning setup. We also present the impact of *non-uniform* tuning on query runtime, and compare both approaches.

5.1. Non-uniform Tuning Methodology

As of today, there is no mechanism in Hive for applying tuning setups at the level of individual jobs within the same query plan, even though MapReduce jobs can be configured individually when submitted separately to Hadoop. We thus extended the Hive query processing engine to switch tuning setups between jobs, executing each job with its own tuning setup. Our extension does not change the current query processing workflow, described in Section 2.1. Instead, it consults with the tuning advisor and rewrites the configuration of every job right before it is queued in Hadoop for execution. Our fork of Hive is fully functional, to the point where a Hive administrator can enable our extension by merely setting a single, new parameter (`hive.optimize.selftuning`) to true. The implementation effort required was modest and amounted to adding $5k$ lines of Java code to Hive.

Figure 5(b) shows the high-level workflow of the non-uniform tuning approach. When a query is submitted (say query 7), it is processed and optimized by the Hive query processing engine: All its jobs have been created and all Hive optimizations have been performed with the query plan looking as depicted in Figure 1. At this point, our extension calls Starfish to profile and optimize every job of query 7 (recall Section 2.2), and stores the recommended tuning setups in a cache repository. The cache associates the *code signature* of each job with its recommended tuning setup [16]. A code sig-

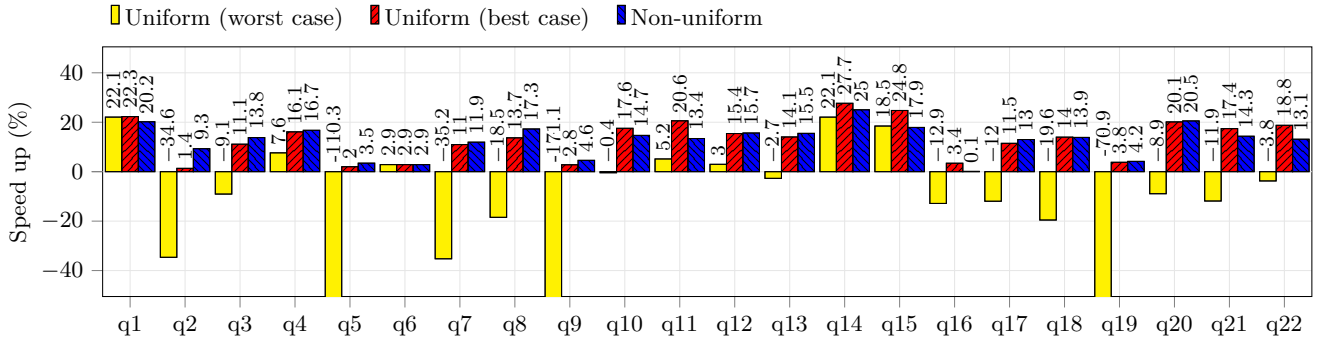


Figure 6: Speedup of uniform and non-uniform tuning approaches relative to the default configuration.

nature is a set of annotations generated during query compilation that capture the query operators executed as part of a job, along with some basic data properties (similar to the information contained in Figure 1). The code signature can be used to uniquely identify a job within the same query plan. When query 7 is submitted again, the code signature of each job is used as a key to look up the corresponding tuning setup from the cache and apply it to the job. In case the dataset size changes, Starfish supports an online optimization mode that takes the new data size into account and generates a new tuning setup on the fly. Hence, each job in the query is submitted with its own optimized tuning setup.

5.2. Uniform vs. Non-Uniform Performance

The uniform and non-uniform tuning approaches can both optimize queries, however, with different impact on performance. Figure 6 presents the speedups for both approaches, including the best and worst cases of uniform tuning. On the one hand, the best case of uniform tuning optimizes queries up to 27.7% (query 14) and does not degrade the performance of any query. The best case of uniform tuning optimizes the runtime of the entire TPC-H workload by 12.1%. Non-uniform tuning achieves a similar overall speedup of 12.2% for the entire TPC-H workload, and is as good as the best case of uniform tuning. On the other hand, the worst case of uniform tuning degrades the performance of 14 out of the 22 queries, with severe slowdowns for queries 5, 9, and 19. The worst case of uniform tuning degrades the performance of the overall TPC-H workload by 40.8%. Hence, non-uniform tuning presents a great and *robust* advantage over uniform tuning.

To further understand the impact of non-uniform tuning on query performance and compare it to uniform tuning, let us consider query 7 again. The best case of uniform tuning applies the recommended parameters of job-2 (see Table 1) to all jobs in query 7 and achieves an 11% speedup. The non-uniform tuning approach assigns each job its very own tuning setup and thereby achieves a 12% speedup. Even though the difference in speedup is small, there are significant advantages to non-uniform tuning with regards to improved resource utilization, as we will see shortly. The increased performance is attributed mainly to the additional speedup achieved for jobs 4 and 5. In particular, the higher values for parameters `mapred.job.reduce.input.buffer.percent` and `mapred.job.shuffle.input.buffer.percent` enable the reduce tasks to buffer more intermediate data in memory, spilling less data to disk, and thereby reducing the amount of both write and read disk I/O performed. In fact, the tuning setup for job-4 completely avoids data spills on the reduce side and *eliminates a total of 15.1 GB of local disk I/O*. As for the remaining jobs, 1 and 6 are short-running and unaffected by the particular choices of parameters. Job 2 is executed with the same settings in both cases, while the performance of job 3 is similar in both cases.

Overall, non-uniform tuning avoids degrading performance by allocating job-specific resources, while the uniform tuning allocates the same amount of resources to all jobs in the query plan. In the strategy of tuning the dominant job, the uniform tuning loses its effectiveness when queries have more than one job dominating query execution. As we have seen in Section 4.3, uniform tuning is effective for optimizing queries in case (A), and ineffective on queries in cases (B) and (C). Non-uniform tuning, however, is effective in all three cases. Another advantage of non-uniform tuning is the *self-tuning nature* of the approach. Human intervention becomes impractical when ordinary OLAP workloads generate tens to hundreds of jobs and many more possible tuning setups.

In our experiments, optimizing jobs individually (non-uniform tuning) always leads to better performance but not always the best. As we can observe from Figure 6, the best-case of uniform approach is a little better (0.1–7.2% higher speedup) than the non-uniform one for 9 out of the 22 TPC-H queries. There are two main reasons for this behavior. First, almost all Hadoop tuning advisors (including Starfish) treat the Map and Reduce functions as black boxes and make simplifying modeling assumptions. For example, some make the propor-

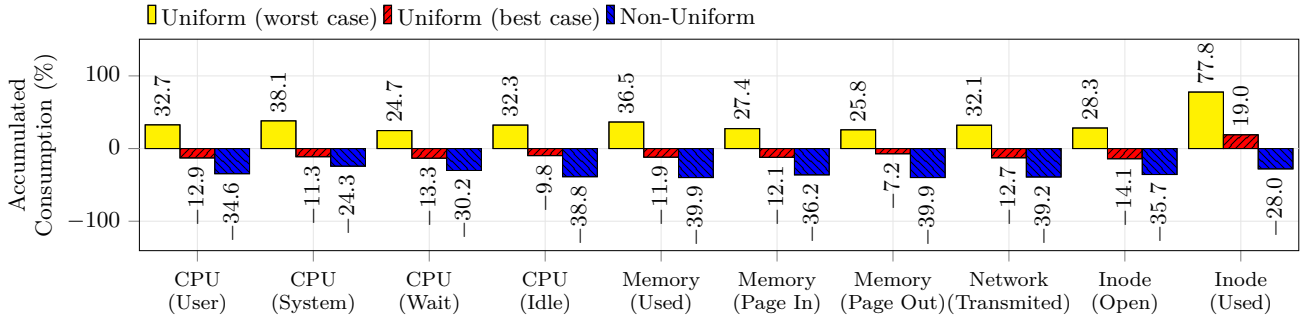


Figure 7: The accumulated resource consumption percentage of uniform and non-uniform tuning relative to the default configuration (baseline). Less is better.

tionality assumption [19], based on which the execution time of a function will double if its input size is doubled. This assumption may hold for simple jobs like WordCount or Sort, but it is not true for jobs that contain multiple relational algebra operators like joins and aggregators. Hence, the modeling, and consequently the tuning recommendations, might not be optimal. Second, performance dependencies between jobs complicate the task of a tuning advisor. For example, setting the number of reduce tasks or enabling output compression for one MapReduce job will affect the performance of the subsequent job as it will affect the number of Map tasks and the need for decompression for the second job, respectively. Hence, it is harder to generate optimal tuning setups for later jobs in a query execution plan. Overall, there is a dire need for: (1) better modeling techniques for MapReduce jobs generated by SQL-on-Hadoop systems, especially since there is inside knowledge of which operations the jobs will perform, (2) better optimizers that can take into consideration the interdependencies of jobs and holistically optimize the entire workflow.

6. Impact on Resource Utilization

Apart from execution time speedup, Hadoop parameter tuning can have a significant impact on the physical resources that are consumed during the execution of HiveQL queries. In this section, we study and compare the impact of uniform and non-uniform tuning on resource utilization.

6.1. Experimental Methodology

During our experiments, we employed the *Collectl* tool[8] to track the consumption of CPU, memory, network, and disk. We monitored the resource consumption of each query from its submission until its completion. *Collectl* runs a light-weight monitoring service on each machine of the cluster, and reads runtime system information from the Linux's `/proc` virtual file system with negligible overhead[9]. When required in our analysis, we also use information from the Hadoop job counters, which are automatically collected by the Hadoop framework during the execution of MapReduce jobs.

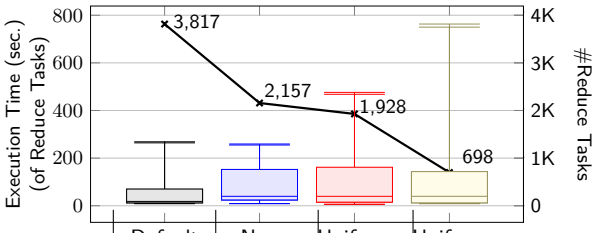
In order to compute the overall resource consumption of the TPC-H workload (to ease our analysis), we: (1) calculate the average consumption per second for each resource from the aggregated values collected on all machines in the cluster; and (2) sum up the averages for each resource, for all queries, to produce the *accumulated consumption*. We compare the resource usage of uniform and non-uniform tuning in Section 6.2 and provide an in-depth analysis of query 7 in Section 6.3.

6.2. Uniform vs. Non-uniform Tuning

Figure 7 depicts the relative percentage of the accumulated resource consumption of the TPC-H workload running with uniform and non-uniform tuning, compared against the default configuration (our baseline). Overall, *non-uniform tuning consumes significantly less computing resources across all relevant metrics*. For instance, non-uniform tuning leads to 35% less CPU utilization and 40% less memory usage than the baseline, whereas the best case of uniform tuning leads to 13% and 12% reductions, respectively. To better understand these results, we discuss next the effect of the number of MapReduce tasks as well as other relevant parameters to the execution behavior of the TPC-H queries.

Under default configuration, the 22 TPC-H queries are executed with 107 MapReduce jobs comprising 16,475 map and 3,817 reduce tasks (yielding 20,292 tasks in total). The number of map tasks is determined by the input data size and typically equals the number of input blocks. Consequently, both the uniform and the non-uniform tuning create almost the same number of map tasks as the default configuration, differing by less than 1%. The number of reduce tasks, however, varies considerably among the different approaches (as can be seen on Figure 8), revealing an interesting trade-off: On the one hand, if the number is set too high, the many short-running reduce tasks increase the scheduling and launching overheads. On the other hand, if the number is set too low, the tasks fail to exploit the potential for cluster parallelization. Rather, each reduce task will

[8] http://collectl.sourceforge.net/
[9] http://collectl.sourceforge.net/Performance.html



	Default	Non Uniform	Uniform (best)	Uniform (worst)
Average	53.79	81.37	88.55	109.21
Std.Dev.	64.97	76.39	95.12	167.33
Range	255.11	246.10	462.17	2,136.51
Max	264.11	255.10	468.17	2,145.51

Figure 8: Distribution of execution time and counts of reduce tasks for each tuning approach. High number of tasks leads to low execution time per task but higher overheads. Low number of tasks leads to low degree of parallelism and cluster utilization. Both lead to overall higher query run times.

need to process a large amount of intermediate data that will probably not fit in the memory buffers. This leads to an increased number of disk spills. Therefore, setting the appropriate number of reduce tasks for each job (along with a few other important parameters) is crucial for a balanced resource utilization and overall performance.

Figure 8 presents the distribution of the execution time of the reduce tasks for each tuning approach. First, we observe the (expected) inverse relationship between the average execution time and the number of reduce tasks. Interestingly, the non-uniform tuning and the best case of uniform tuning decrease the number of reduce tasks by 1.8× and 2.0×, respectively, but each task on average takes only 1.5× and 1.6× more time to complete, compared to the default configuration. Nonetheless, the most important difference is the variability in task execution time. Both uniform tuning approaches exhibit stretched distributions, with worst-case outliers that are almost 20×larger than the corresponding average execution time. Non-uniform tuning, on the other hand, *leads to the smallest range of execution time values, showcasing once again the robustness of this method.* Finally, the default configuration uses a large number of reduce tasks with shorter execution times (and naturally low variability). Even though the execution times of the individual reduce tasks are shorter, there are more tasks and their management overheads increase the execution time of the overall job. Thus, the actual query execution times are generally inferior to non-uniform tuning, as shown in Figure 6 and also discussed in Section 5.2.

Another positive artifact of reduced execution time variability is the reduction or elimination of *straggler tasks*, i.e., tasks that make slower progress compared to other tasks. When Hadoop detects a straggler task, it will run a *speculative copy* of that task on another node to finish computation faster. As soon as one of the two tasks completes, the other one is killed. In one execution run of the TPC-H queries with the default configuration, we observed that 191 reduce tasks were killed. This implies that there are 5% more reduce tasks than needed that are being launched and scheduled[10]. Non-uniform tuning and the best case of uniform tuning decrease the number of failed reduce tasks to less than 1%, thereby *reducing the number of straggler tasks by 5×*.

In addition to setting the number of reduce tasks correctly, several other parameters (e.g., `mapred.job.reduce.input.buffer.percent` and `mapred.job.shuffle.input.buffer.percent`) can regulate the amount of memory used to buffer map outputs during shuffle and reduce phases. When these buffers fill up, Hadoop starts to write the map outputs to the local file system. The amount of memory given to map and reduce tasks directly impacts the amount of data spills (on the map or reduce side, respectively). Note that the data written between jobs in a query plan and the final results of the queries are written to the distributed file system and are not affected by parameter settings. Figure 9 presents the accumulated data written to the local and distributed file systems during the shuffle and reduce phases. Considering the total amount of data materialized to the local file system, *non-uniform tuning writes 5.3× (584.4GB) less than the default configuration.* In fact, the non-uniform tuning completely eliminates data spills on the reduce side from 92 out of the 107 jobs generated from the TPC-H, which yields 4.2× more jobs without data spills on the reduce side than the default configuration. The best-case of uniform tuning writes 1.9× (340.7GB) less data, while the worst-case of uniform tuning writes only 8.8% (63.4GB) less.

As a consequence of writing less data to the local file system, non-uniform tuning is more parsimonious with other resources as well (see Figure 7). For instance, setting job buffers with non-uniform tuning (e.g., `io.sort.mb`, `mapred.job.reduce.input.buffer.percent`) leads to 39.9% fewer page outs, and, consequently, the CPU waits 30.2% less. The decrease in memory page outs as well as the reduced data spills are instrumental in reducing the number of open files by 35.7%. Other parameters such as `io.sort.spill.percent` and `mapred.job.shuffle.merge.percent`, when properly set, enable better overlapping between CPU processing and I/O, contributing to lower CPU waits.

6.3. In-Depth Analysis for Query 7

Figure 10 presents the utilization of CPU, memory, and network for query 7. Table 3 further presents the average run time for each job of query 7, during its execution with the default configuration, non-uniform

[10] This effect is consistently reproducible across repeated runs. Across our three runs, we observed an average number of failed reduce tasks of 185.33, and a standard deviation of 6.03.

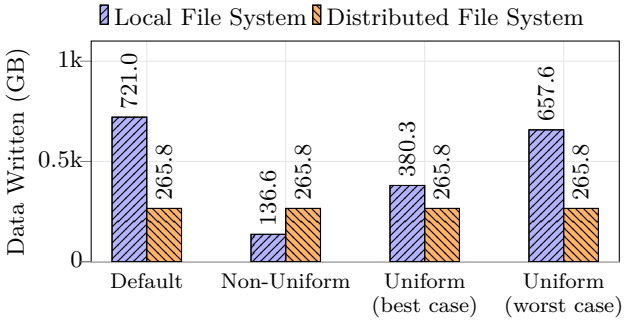


Figure 9: The total amount of data written to the local and distributed file system during the shuffle and reduce phases of the 22 TPC-H queries.

tuning, and best and worst cases of uniform tuning. In Figure 10, solid black lines indicate the beginning of each job in the sequential query plan schedule, and dashed lines indicate the end of query execution. The submission of the jobs follows a sequential order derived from the query plan in Figure 1. The data presented is an average of the data from 10 machines in the cluster, measured during a single execution. Note that job-1 is a small job that executes in only about 6 seconds across all configurations (and hence almost not visible in Figure 10), because it is a map-only job executing on the small table *nations*. Similarly, job-6 is also a small job that materializes the query results to the final table, running in 12 seconds on average across all configurations. In our discussion below, we will focus on jobs 2, 3, and 4, which perform one join operation each and together constitute more than 90% of the total execution time of query 7.

We first analyze the effect of the tuning approaches to CPU utilization. Consider job-2, which filters table *orders* (34GB) and scans table *lineitem* (150GB) during the map phase, and performs a join during the reduce phase. Our first observation from Figure 10 is that the default configuration and the worst case of uniform tuning exhibit low CPU utilization during the first half of the job (i.e., the map phase), whereas the opposite is true for the non-uniform tuning and the best case of uniform tuning. The explanation is traced to the settings of `io.sort.mb`, which determines the map output buffer size, and `io.sort.spill.percent`, which sets a threshold for when to sort and spill the buffered data (see Table 1). The small buffer size (100MB and 547MB in default configuration and worse case of uniform tuning, respectively) in combination with the small threshold (0.51) in the worse case of uniform tuning, leads job-2 to sort and spill small chunks of data more frequently during the map phase, which, consequently, induces low CPU utilization.

Another interesting observation is the lower variation in CPU utilization for non-uniform tuning compared to the other cases, especially for job-4. In the best case of uniform tuning, the low values for `io.sort.spill.percent`, `mapred.inmem.merge.threshold`, and `mapred.job.shuffle.merge.percent` cause frequent rounds of sort-spill-merge operations, leading to variability in CPU usage. The settings used in the non-uniform tuning case, on the other hand, *achieve a better overlap between the CPU processing and I/O spills*, leading to higher CPU utilization and to a lower job run time.

Let us consider memory utilization next, also shown in Figure 10. Our first observation is the low memory usage of job-2 during default configuration, which is attributed to the low `io.sort.mb` setting (100MB). The map functions of job-2 perform only scan and filter operations, which do not require much execution memory. Even though there is a lot of available memory, only 100MB can be used by each map task for buffering map output data, leading to a significant underutilization of memory. On the contrary, the high `io.sort.mb` settings used by the other approaches (ranging from 547MB to 1050MB) enable map tasks to buffer more output data and better utilize memory.

The memory consumption of job-3 in the best case of uniform tuning experiences a noticeable drop during the reduce phase. The low setting of `mapred.reduce.input.buffer.percent` at 0.17 means that only a small percentage of memory is used to buffer map output data during the reduce execution. In addition, job-3, which joins the table *customer* (4.6GB) with the output of job-2 (20GB), uses 36 reduce tasks. Hence, each reduce task processes approximately 0.6GB of data. The combined effect of the two aforementioned factors contribute to the reduced memory usage of job-3.

Finally, let us now observe network consumption. Recall that Figure 10 shows an average of data from 10 machines, so when one machine is transmitting data, another machine is receiving it. Thus, the amount of data transmitted and received through the network at a given point in time are congruent. Network consumption may be affected by many factors, including how the data is distributed across the nodes of the cluster, how balanced the tasks are scheduled to each node, and the replication degree of data. Hence, individual configuration parameters have a lower impact on network utilization compared to other resources. Nonetheless, the overall settings used by non-uniform tuning lead to a smoother network consumption compared to other approaches, which occasionally suffer from big network spikes.

7. Lessons Learned & Conclusion

While there has been a significant amount of research on automatic parameter tuning for distributed computing platforms such as Hadoop and Spark (e.g., [22, 37, 46, 56, 4]), there has been very little work on (i) how these parameters affect the execution and performance of SQL-on-Hadoop systems, and (ii) how to tune

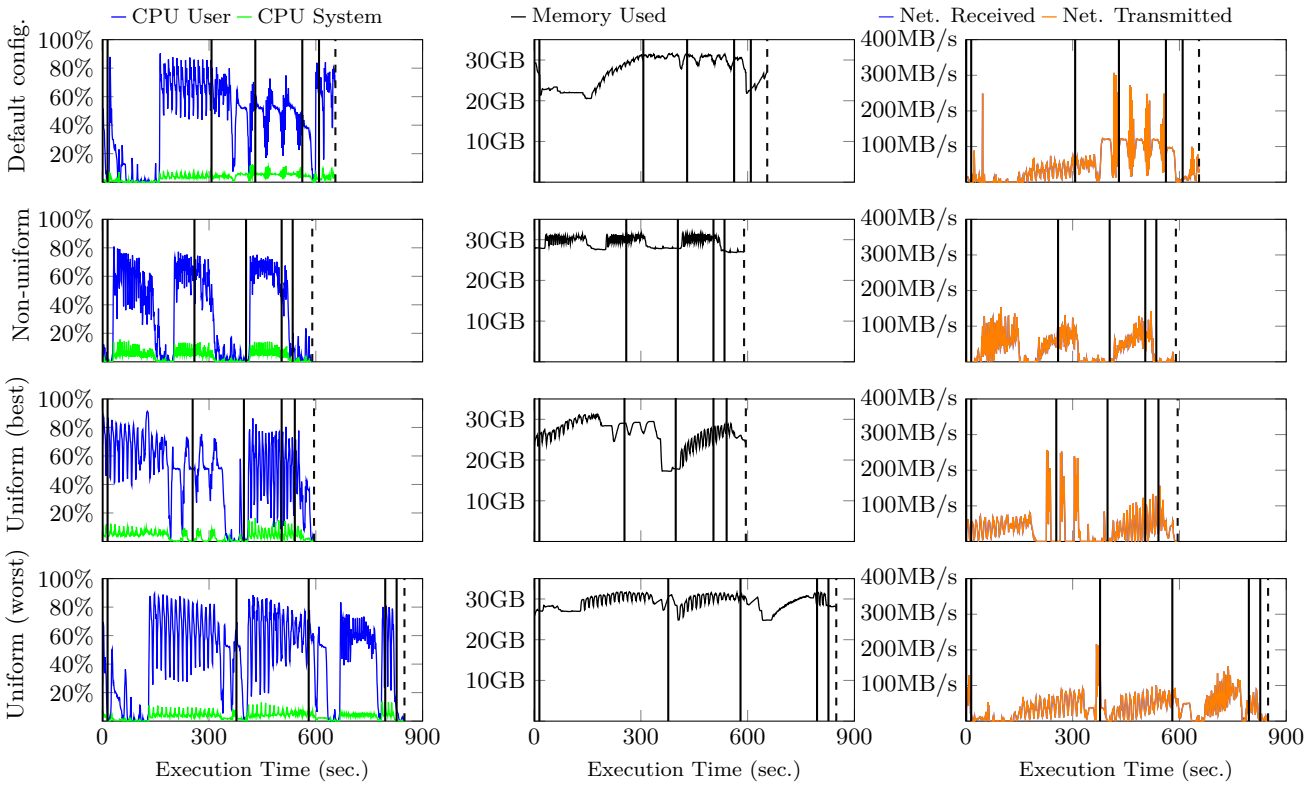


Figure 10: Average resource consumption of the 10 machines of the cluster during a single execution of TPC-H query 7. Solid black lines indicate the beginning of each of the 6 jobs in the query plan, and dashed black lines indicate the end of query execution.

these parameters within the context of SQL-on-Hadoop workloads. In this paper, we attempt to bridge this gap by studying how current Hadoop tuning advisors can be employed for optimizing the performance of the popular SQL-on-Hadoop engine Hive. The major lessons learned from our experimental study are:

1. Parameter tuning is important. Tuning the underlying Hadoop parameters can have a significant impact on the execution time and the resource utilization of queries executing on SQL-on-Hadoop systems. In particular, our results reveal that different parameter settings can cause run time variations between -171% (i.e., 2.8× slower) and 25% speedup over default settings. Also, different settings can influence resource utilization and lead to better CPU utilization, improved memory usage, reduced disk usage, and more evenly distributed network utilization (i.e., fewer spikes).

Reduced resource consumption can have major benefits for customers and providers of Infrastructure-as-a-Service products. From the customer's perspective, non-uniform tuning allows for more efficient resource consumption, which directly translates to lower charges depending on the pricing model. For instance, storage costs in Amazon AWS Elastic Block Store (EBS) are $0.10/GB for general purpose (gp2) volumes. The amount of data spilled to disk for intermediate data, as depicted by Figure 9, will be directly reflected in billable costs. With reduced data spills, customers could also provision for lower (and cheaper) I/O per second (IOPS) rates. In addition, lower resource utilization allows for increased query throughput because more queries can be executed in the same amount of time, potentially reducing the cost of cloud deployments. From the provider's perspective, non-uniform tuning has the potential to reduce infrastructure costs, such as air cooling with more efficient disk usage and less data movement, and to increase machine throughput with less resource consumption and more concurrent clients.

Table 3
Average run time (in seconds) per job of query 7 with the default configuration, non-uniform, and uniform tuning.

	Default config.	Non-uniform	Uniform best-case	Uniform worst-case
job-1	6.0	6.0	5.5	5.0
job-2	300.0	252.0	248.0	371.5
job-3	123.7	145.0	144.0	203.0
job-4	132.0	100.0	105.5	214.5
job-5	46.3	31.0	37.0	32.5
job-6	12.0	12.0	12.0	12.0
Total	620.0	546.0	552.0	838.5

2. Uniform tuning is problematic. The current practice of using one tuning setup per query suffers from two main limitations. First, finding a tuning setup that equally benefits all jobs is difficult, if

not impossible, since current Hadoop tuning advisors are designed for tuning one MapReduce job at-a-time. Hence, even though a particular tuning setup can speed up one job significantly, it may damage other jobs in the same query, leading to a sub-optimal speedup or even an overall slowdown. Second, the process of selecting a good tuning setup is time-consuming, as it typically relies on trial-and-error. Even intuitive strategies such as tuning for the dominant job (i.e., the longest running job) are not always effective and can actually slow down a query.

3. Non-uniform tuning is robust. Tuning each job in a query independently has been shown to provide run time speedups that are similar to the best-observed speedups in our experiments. Unlike uniform tuning, the non-uniform tuning approach does not suffer from any serious run time variations and, consequently, reduces the need for speculative execution. Most importantly, the non-uniform tuning leads to better resource utilization. For instance, both CPU usage and memory paging were reduced by over 40%, while the total amount of data written to and read from the local file system was reduced by 5× when compared to default settings. The aforementioned benefits are not bound to Starfish but rather are a consequence of the non-uniform tuning methodology that optimizes each job independently. We are confident that any Hadoop tuning advisor could be used in place of Starfish, with similar results. Moreover, non-uniform tuning enables plugging other types of tuning advisors that optimize for specific resources (e.g., disk I/O) or energy rather than execution time, as those metrics might be more important for some public or hybrid cloud scenarios.

4. Challenges apply to other SQL-on-Hadoop systems. Even though we focus on tuning Hadoop parameters for Hive queries, we believe that our results and conclusions generalize to other SQL-on-Hadoop systems as many of them fork the Hive project, like Shark, or share some key architectural and processing design choices. First, such systems accept SQL-like queries and break them down to individual jobs for execution on the underlying framework (e.g., Hadoop, Tez, Spark). These jobs are all affected by parameters that control the degree of parallelism for tasks, various memory settings, etc. In addition, SQL-on-Hadoop queries have a single point of configuration (e.g., command line, query code), that makes it impossible for developers to manually determine and set a specific tuning setup per job. Even with the help of tuning advisors, this single point of configuration will force all jobs in the query plan to share the same tuning setup, which can lead to performance degradation.

The issues with uniform tuning apply to other types of systems as well, like Impala, Drill, and Spark. For instance, Impala distributes a query for processing among the available cluster nodes in a way that resembles jobs and tasks. Impala updates system health information across all the nodes through a "statestore" component (in Impala terminology) and keeps all changes in the metadata of the SQL statements in "catalog" daemons. The non-uniform approach would leverage both daemons to swap tuning parameters according to the processing needs.

In Spark, a job consists of a directed acyclic graph of stages, where each stage comprises a collection of tasks. Yet, all parameters that affect resource configuration and scheduling (such as number of cores to use, memory sizes, etc.) are set at the level of jobs and apply uniformly to all stages and tasks. This problem has already been identified and efforts are made towards setting task resource requirements and configuration at the stage level [52]. Currently, Spark tuning advisors such as [56, 48, 4] recommend parameters only at the level of individual jobs.

5. New research opportunities await. Most Hadoop tuning advisors (e.g., [35, 37, 13]), including Starfish, treat the Map and Reduce functions as black-boxes. However, when the MapReduce jobs are generated from SQL-on-Hadoop engines, the information regarding the actual operators comprising the tasks is available. Such information can significantly improve the performance modeling employed by the tuning advisors, which in turn can improve the tuning setup recommendations. At the same time, current tuning advisors are designed for optimizing each job in isolation. However, SQL-on-Hadoop engines typically generate directed acyclic graphs of MapReduce jobs, with various inter-dependencies, as discussed in Section 5.2. Hence, a new line of tuning advisors that will incorporate query-specific modeling and workflow-aware tuning is necessary for catering to the specific requirements of SQL-on-Hadoop engines. Another interesting research direction would be to push physical tuning decisions into the cost optimizer of an SQL-on-Hadoop engine. At optimization time, there is access to various statistics such as cardinality estimates and possible access paths, which can be used to (i) improve the effectiveness of the tuning choices, and to (ii) apply the selected tuning choices to the individual jobs (i.e., in a non-uniform way). The push towards non-uniform tuning also implies a push towards a fully automated tuning solution. Otherwise, it would be hard, if not impossible, for an administrator to manually specify parameters for individual jobs that are generated during a query execution. Overall, as the clusters are growing in size and the workloads are becoming more complex, it becomes ever more essential for SQL-on-Hadoop systems to offer self-managing features such as automatic performance tuning.

With this experimental study, we share our observations with the systems research community. We hope to create an awareness for this problem as well as to initiate new research on automatic parameter tuning for SQL-on-Hadoop systems.

Table 4
Hadoop tuning advisors along with the reported speedup against default configuration, the search strategy employed, and the workloads used for evaluation.

			Workloads																						
Tuning Advisor	Speedup	Search Strategy	Adjacent list	Biagram	Grep	Histogram	Inverted Index	K-Core	K-Means	LinkGraph	Nutch	PageRank	Pi Estimation	Sort	TeraSort	Text Classification	Weakly Conn. Comp.	Word Co-Occurrence	WordCount	Custom Workload	Hive Aggregation	Join	Order By	TPC-DS (Subset)	TPC-H (Subset)
Starfish [19, 22]	13.9x	Random Recursive Search								✓					✓			✓	✓			✓			
Predator [57]	5x	Grid Hill Climbing													✓				✓						
Panacea [38]	3x	Exhaustive Search						✓	✓			✓					✓		✓						
SVR [60]	3x	Exhaustive Search												✓					✓						
PPABS [58]	38.40%	Simulated Annealing			✓										✓				✓						
Gunther [35]	33.00%	Genetic Algorithm							✓		✓			✓	✓										
AutoTune [62]	-	-																		✓					✓
MRTuner [46]	4.41x	Producer, Transporter, Consumer										✓			✓	✓									
MROnline [34]	30.00%	Smart Hill Climbing		✓	✓		✓						✓		✓				✓						
AACT [33]	10x	Exhaustive Search											✓		✓				✓						
JellyFish [13]	74.00%	Hill Climbing			✓	✓	✓								✓	✓			✓						
Chen [8]	8x	Stochastic Hill Climbing													✓			✓	✓		✓				
MR-COF [37]	35.00%	Genetic Algorithm			✓									✓					✓						
RFHOC [6]	7.4x	Genetic Algorithm	✓				✓							✓	✓				✓						
Zhang [61]	40.00%	Double-Threshold Heuristic												✓	✓				✓						
Lee [31]	29.00%	Fuzzy Inference			✓										✓				✓						
Khan [28]	71.00%	Particle Swarm Optimization												✓					✓						
Kumar [30]	66.00%	Stochastic Approximation		✓	✓		✓								✓			✓						✓	
Jain [25]	38.51%	Exhaustive Search																	✓						
MEST [5]	-	Genetic Algorithm							✓			✓		✓	✓				✓						
Prasad [12]	-	Exhaustive Search																		✓				✓	
mrEtalon [7]	30.00%	Simulated Annealing		✓			✓			✓			✓	✓	✓			✓	✓			✓	✓		
Khaleel [27]	73.39%	Genetic Algorithm													✓				✓						

8. Acknowledgments

This work was supported by the AWS Cloud Credits for Research program and CAPES Brazil.

A. Appendix A

Table 4 lists several Hadoop tuning advisors along with their reported speedups, employed search strategies, and the workloads used for evaluation. All reported speedups are against run times using Hadoop's default configuration (i.e., our baseline). Tables 5 and 6 list selected parameter values generated by Starfish for each MapReduce job compiled from the TPC-H queries.

Table 5
Selected parameter values for each MapReduce job of TPC-H queries 1-11.

Query	Job Id	io.sort.mb	io.sort.factor	io.sort.record.percent	io.sort.spill.percent	mapred.inmem.merge.threshold	mapred.job.reduce.input.buffer.percent	mapred.job.shuffle.input.buffer.percent	mapred.job.shuffle.merge.percent	mapred.reduce.tasks
Query-1	1	568	100	0.090	0.776	675	0.035	0.638	0.853	8
	2	100	10	0.050	0.800	1000	0.000	0.700	0.660	1
Query-2	1	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	2	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	3	655	82	0.010	0.683	110	0.261	0.490	0.468	25
	4	668	57	0.016	0.574	969	0.441	0.461	0.475	25
	5	1126	53	0.351	0.716	928	0.772	0.278	0.420	8
	6	758	76	0.500	0.726	916	0.181	0.560	0.203	8
Query-3	1	809	42	0.011	0.526	818	0.124	0.515	0.416	24
	2	618	97	0.011	0.665	710	0.726	0.721	0.778	36
	3	1212	24	0.012	0.528	558	0.065	0.657	0.424	8
	4	818	16	0.012	0.589	673	0.759	0.375	0.749	8
Query-4	1	579	25	0.015	0.548	641	0.697	0.590	0.730	23
	2	1254	34	0.010	0.514	163	0.716	0.726	0.613	25
	3	572	93	0.024	0.742	222	0.101	0.482	0.611	8
	4	914	15	0.188	0.841	952	0.318	0.550	0.367	8
Query-5	1	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	2	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	3	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	4	959	39	0.011	0.747	936	0.712	0.309	0.855	30
	5	715	76	0.010	0.534	432	0.232	0.553	0.545	21
	6	1131	42	0.431	0.673	727	0.583	0.212	0.563	8
	7	1239	99	0.176	0.811	615	0.740	0.621	0.833	8
Query-6	1	874	75	0.402	0.719	758	0.335	0.550	0.738	8
Query-7	1	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	2	879	32	0.015	0.605	470	0.174	0.299	0.675	36
	3	593	64	0.010	0.785	651	0.304	0.381	0.818	36
	4	593	64	0.010	0.785	651	0.304	0.381	0.818	36
	5	1050	71	0.405	0.638	698	0.441	0.397	0.479	8
	6	547	22	0.230	0.510	568	0.240	0.510	0.510	8
Query-8	1	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	2	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	3	665	56	0.013	0.553	734	0.147	0.544	0.418	23
	4	931	47	0.011	0.576	974	0.680	0.543	0.806	36
	5	741	33	0.011	0.527	258	0.154	0.697	0.627	22
	6	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	7	910	54	0.172	0.742	594	0.444	0.615	0.563	8
	8	1230	41	0.262	0.582	241	0.659	0.604	0.270	8
Query-9	1	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	2	555	67	0.011	0.682	750	0.318	0.770	0.809	180
	3	555	67	0.011	0.682	750	0.318	0.770	0.809	180
	4	616	67	0.010	0.571	912	0.433	0.775	0.632	144
	5	569	25	0.012	0.714	713	0.563	0.817	0.599	36
	6	1107	18	0.166	0.780	744	0.129	0.896	0.243	8
	7	1038	30	0.034	0.672	648	0.664	0.282	0.702	8
Query-10	1	1220	36	0.358	0.697	568	0.580	0.737	0.633	25
	2	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	3	587	83	0.011	0.677	621	0.576	0.605	0.382	36
	4	632	99	0.012	0.500	848	0.295	0.400	0.349	20
	5	765	40	0.010	0.706	325	0.000	0.637	0.727	21
Query-11	1	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	2	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	3	618	42	0.013	0.506	675	0.091	0.300	0.622	8
	4	618	42	0.013	0.506	675	0.091	0.300	0.622	8
	5	1292	68	0.325	0.873	962	0.170	0.777	0.591	3

Table 6
Selected parameter values for each MapReduce job of TPC-H queries 12-22.

Query	Job Id	io.sort.mb	io.sort.factor	io.sort.record.percent	io.sort.spill.percent	mapred.inmem.merge.threshold	mapred.job.reduce.input.buffer.percent	mapred.job.shuffle.input.buffer.percent	mapred.job.shuffle.merge.percent	mapred.reduce.tasks
Query-12	1	592	42	0.016	0.646	925	0.403	0.713	0.357	28
	2	100	10	0.050	0.800	1000	0.000	0.700	0.660	1
	3	1028	40	0.332	0.832	211	0.047	0.394	0.325	7
Query-13	1	551	21	0.011	0.729	332	0.389	0.665	0.709	27
	2	604	39	0.016	0.502	215	0.363	0.723	0.874	8
	3	1279	62	0.489	0.692	314	0.461	0.838	0.717	8
	4	946	11	0.107	0.706	908	0.593	0.618	0.277	8
Query-14	1	645	48	0.455	0.662	673	0.293	0.348	0.585	21
	2	100	10	0.050	0.800	1000	0.000	0.700	0.660	1
Query-15	1	745	63	0.257	0.885	745	0.610	0.362	0.230	8
	2	669	16	0.223	0.696	30	0.156	0.252	0.827	8
	3	516	48	0.011	0.713	964	0.639	0.513	0.617	8
	4	532	84	0.334	0.845	36	0.614	0.343	0.427	2
Query-16	1	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	2	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	3	538	81	0.011	0.695	703	0.644	0.535	0.867	25
	4	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	5	520	31	0.016	0.515	289	0.144	0.896	0.688	20
	6	1261	98	0.014	0.515	952	0.316	0.571	0.318	8
Query-17	1	547	78	0.010	0.678	913	0.606	0.724	0.594	72
	2	543	57	0.011	0.560	840	0.172	0.740	0.689	72
	3	745	57	0.010	0.550	786	0.251	0.471	0.482	8
	4	830	56	0.134	0.776	88	0.288	0.634	0.519	7
Query-18	1	549	26	0.024	0.751	964	0.250	0.869	0.520	27
	2	624	97	0.012	0.731	752	0.486	0.663	0.414	36
	3	644	79	0.013	0.544	707	0.510	0.607	0.775	72
	4	539	59	0.129	0.848	525	0.011	0.419	0.406	8
	5	622	87	0.125	0.521	473	0.309	0.706	0.824	8
Query-19	1	612	88	0.012	0.574	405	0.263	0.452	0.898	144
	2	1192	27	0.026	0.539	24	0.130	0.495	0.601	8
Query-20	1	980	50	0.463	0.890	346	0.244	0.265	0.648	8
	2	856	47	0.011	0.550	888	0.341	0.568	0.484	25
	3	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	4	555	67	0.013	0.594	704	0.288	0.561	0.501	21
	5	638	49	0.012	0.665	863	0.490	0.439	0.442	8
	6	1256	94	0.224	0.579	536	0.273	0.306	0.407	8
Query-21	1	513	29	0.013	0.632	776	0.794	0.571	0.508	72
	2	647	61	0.015	0.527	798	0.702	0.529	0.854	36
	3	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	4	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	5	714	25	0.011	0.626	483	0.391	0.422	0.341	23
	6	539	81	0.012	0.596	777	0.169	0.414	0.568	28
	7	633	86	0.015	0.549	460	0.359	0.865	0.856	25
	8	1209	47	0.423	0.614	550	0.590	0.599	0.599	8
	9	1260	51	0.138	0.736	70	0.000	0.370	0.277	8
Query-22	1	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	2	100	10	0.050	0.800	1000	0.000	0.700	0.660	0
	3	1297	82	0.273	0.654	877	0.053	0.338	0.859	8
	4	665	24	0.012	0.748	234	0.000	0.347	0.896	24
	5	605	40	0.011	0.561	384	0.200	0.334	0.294	8
	6	861	68	0.321	0.656	854	0.750	0.237	0.719	8
	7	726	29	0.443	0.509	458	0.591	0.709	0.295	8

References

[1] Afrati, F., Dolev, S., Korach, E., Sharma, S., Ullman, J.D., 2016. Assignment Problems of Different-Sized Inputs in MapReduce. ACM Trans. Knowl. Discov. Data 11, 18:1–18:35. URL: http://doi.acm.org/10.1145/2987376, doi:10.1145/2987376.

[2] Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., Zaharia, M., 2015. Spark SQL: Relational Data Processing in Spark, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, ACM. pp. 1383–1394. doi:10.1145/2723372.2742797.

[3] Babu, S., 2010. Towards Automatic Optimization of MapReduce Programs, in: Proceedings of the 1st ACM Symposium on Cloud Computing, ACM. pp. 137–142. doi:10.1145/1807128.1807150.

[4] Bao, L., Liu, X., Chen, W., 2018. Learning-based Automatic Parameter Tuning for Big Data Analytics Frameworks, in: Proc. of the IEEE Intl. Conf. on Big Data, IEEE. pp. 181–190.

[5] Bei, Z., Yu, Z., Liu, Q., Xu, C., Feng, S., Song, S., 2017. MEST: A Model-Driven Efficient Searching Approach for MapReduce Self-Tuning. IEEE Access 5, 3580–3593. doi:10.1109/ACCESS.2017.2672675.

[6] Bei, Z., Yu, Z., Zhang, H., Xiong, W., Xu, C., Eeckhout, L., Feng, S., 2016. RFHOC: A Random-Forest Approach to Auto-Tuning Hadoop's Configuration. IEEE Transactions on Parallel and Distributed Systems 27, 1470–1483. doi:10.1109/TPDS.2015.2449299.

[7] Cai, L., Qi, Y., Li, J., 2017. A Recommendation-Based Parameter Tuning Approach for Hadoop, in: Proceedings of the IEEE 7th International Symposium on Cloud and Service Computing (SC2), pp. 223–230. doi:10.1109/SC2.2017.41.

[8] Chen, C.O., Zhuo, Y.Q., Yeh, C.C., Lin, C.M., Liao, S.W., 2015. Machine Learning-Based Configuration Parameter Tuning on Hadoop System, in: Proceedings of the 2015 IEEE International Congress on Big Data, IEEE Computer Society. pp. 386–392. doi:10.1109/BigDataCongress.2015.64.

[9] Chen, Y., Qin, X., Bian, H., Chen, J., Dong, Z., Du, X., Gao, Y., Liu, D., Lu, J., Zhang, H., 2014. A Study of SQL-on-Hadoop Systems, in: Proceedings of the Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware, Springer. pp. 154–166.

[10] Cherkasova, L., 2011. Performance Modeling in MapReduce Environments: Challenges and Opportunities, in: Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering, ACM. pp. 5–6. doi:10.1145/1958746.1958752.

[11] Chiba, T., Yoshimura, T., Horie, M., Horii, H., 2018. Towards Selecting Best Combination of SQL-on-Hadoop Systems and JVMs, in: Proceedings of the 11th IEEE International Conference on Cloud Computing (CLOUD), pp. 245–252. doi:10.1109/CLOUD.2018.00038.

[12] Deshpande, P.M., Margoor, A., Venkatesh, R., 2018. Automatic Tuning of SQL-on-Hadoop Engines on Cloud Platforms, in: Proceedings of the IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 508–515. doi:10.1109/CLOUD.2018.00071.

[13] Ding, X., Liu, Y., Qian, D., 2015. JellyFish: Online Performance Tuning with Adaptive Configuration and Elastic Container in Hadoop Yarn, in: Proceedings of the IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS), pp. 831–836. doi:10.1109/ICPADS.2015.112.

[14] Drill, 2021. Apache Drill: Schema-free SQL Query Engine for Hadoop, NoSQL and Cloud Storage. https://drill.apache.org/. 2021 [Online; accessed 25-Feb-2021].

[15] Ead, M., Herodotou, H., Aboulnaga, A., Babu, S., 2014. PStorM: Profile Storage and Matching for Feedback-Based Tuning of MapReduce Jobs, in: Proceedings of the 17th Intl. Conf. on Extending Database Technology (EDBT), pp. 1–12.

[16] Filho, E.R.L., de Almeida, E.C., Scherzinger, S., 2019. Don't Tune Twice: Reusing Tuning Setups for SQL-on-Hadoop Queries, in: Conceptual Modeling, Springer International Publishing. pp. 93–107.

[17] Floratou, A., Minhas, U.F., Özcan, F., 2014. SQL-on-Hadoop: Full Circle Back to Shared-nothing Database Architectures. Proc. of VLDB Endowment (PVLDB) 7, 1295–1306. doi:10.14778/2732977.2733002.

[18] Glushkova, D., Jovanovic, P., Abelló, A., 2019. MapReduce Performance Model for Hadoop 2.x. Inf. Syst. 79, 32–43.

[19] Herodotou, H., Babu, S., 2011. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. Proc. of VLDB Endowment (PVLDB) 4, 1111–1122.

[20] Herodotou, H., Babu, S., 2013. A What-if Engine for Cost-based MapReduce Optimization. IEEE Data Eng. Bull. 36, 5–14.

[21] Herodotou, H., Chen, Y., Lu, J., 2020. A Survey on Automatic Parameter Tuning for Big Data Processing Systems. ACM Computing Surveys (CSUR) 53, 1–37.

[22] Herodotou, H., Lim, H., Luo, G., Borisov, N., Dong, L., Cetin, F.B., Babu, S., 2011. Starfish: A Self-tuning System for Big Data Analytics, in: Proceedings of the 5th Biennial Conf. on Innovative Data Systems Research (CIDR), pp. 261–272.

[23] Heudecker, N., Adrian, M., 2015. Survey Analysis: Hadoop Adoption Drivers and Challenges. Gartner, Inc.

[24] Huai, Y., Chauhan, A., Gates, A., Hagleitner, G., Hanson, E.N., O'Malley, O., Pandey, J., Yuan, Y., Lee, R., Zhang, X., 2014. Major Technical Advancements in Apache Hive, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, ACM. pp. 1235–1246. doi:10.1145/2588555.2595630.

[25] Jain, A., Choudhary, M., 2017. Analyzing & optimizing Hadoop performance, in: Proceedings of the 2017 International Conference on Big Data Analytics and Computational Intelligence (ICBDAC), pp. 116–121. doi:10.1109/ICBDACI.2017.8070820.

[26] Jiang, D., Ooi, B.C., Shi, L., Wu, S., 2010. The Performance of MapReduce: An In-depth Study. Proc. of VLDB Endowment (PVLDB) 3, 472–483.

[27] Khaleel, A., Al-Raweshidy, H., 2018. Optimization of Computing and Networking Resources of a Hadoop Cluster Based on Software Defined Network. IEEE Access .

[28] Khan, M., Huang, Z., Li, M., Taylor, G.A., Khan, M., 2017. Optimizing Hadoop parameter settings with gene expression programming guided PSO. Concurrency Computation URL: http://doi.wiley.com/10.1002/cpe.3786, doi:10.1002/cpe.3786.

[29] Kornacker, M., Behm, A., Bittorf, V., Bobrovytsky, T., Ching, C., Choi, A., Erickson, J., Grund, M., Hecht, D., Jacobs, M., et al., 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop., in: Proceedings of the 7th Biennial Conf. on Innovative Data Systems Research (CIDR), p. 9.

[30] Kumar, S., Padakandla, S., Chandrashekar, L., Parihar, P., Gopinath, K., Bhatnagar, S., 2017. Scalable Performance Tuning of Hadoop MapReduce: A Noisy Gradient Approach, in: Proceedings of the IEEE 10th International Conference on Cloud Computing (CLOUD), pp. 375–382.

[31] Lee, G.J., Fortes, J.A.B., 2016. Hadoop Performance Self-Tuning Using a Fuzzy-Prediction Approach, in: Proceedings of the IEEE International Conference on Autonomic Computing (ICAC), pp. 55–64.

[32] Lee, R., Luo, T., Huai, Y., Wang, F., He, Y., Zhang, X., 2011. YSmart: Yet Another SQL-to-MapReduce Translator, in: Proceedings of the 31st International Conference on Distributed Computing Systems, IEEE. pp. 25–36.

[33] Li, C., Zhuang, H., Lu, K., Sun, M., Zhou, J., Dai, D., Zhou, X., 2014. An Adaptive Auto-configuration Tool for Hadoop, in: Proceedings of the 19th International Conference on Engineering of Complex Computer Systems, pp. 69–72.

[34] Li, M., Zeng, L., Meng, S., Tan, J., Zhang, L., Butt, A.R., Fuller, N., 2014. MRONLINE: MapReduce Online Performance Tuning, in: Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing, Association for Computing Machinery. p. 165–176. doi:10.1145/2600212.2600229.

[35] Liao, G., Datta, K., Willke, T.L., 2013. Gunther: Search-based Auto-tuning of MapReduce, in: Proceedings of the European Conference on Parallel Processing, Springer. pp. 406–419.

[36] Lim, H., Herodotou, H., Babu, S., 2012. Stubby: A Transformation-based Optimizer for MapReduce Workflows. Proc. of VLDB Endowment (PVLDB) 5, 1196–1207.

[37] Liu, C., Zeng, D., Yao, H., Hu, C., Yan, X., Fan, Y., 2015. MR-COF: A Genetic MapReduce Configuration Optimization Framework, in: Wang, G., Zomaya, A., Martinez, G., Li, K. (Eds.), Proceedings of Algorithms and Architectures for Parallel Processing, Springer International Publishing. pp. 344–357.

[38] Liu, J., Ravi, N., Chakradhar, S., Kandemir, M., 2012. Panacea: Towards Holistic Optimization of MapReduce Applications, in: Proceedings of the Tenth International Symposium on Code Generation and Optimization, Association for Computing Machinery. p. 33–43. doi:10.1145/2259016.2259022.

[39] Mahgoub, A., Medoff, A., Kumar, R., Mitra, S., Klimovic, A., Chaterji, S., Bagchi, S., 2020. OPTIMUSCLOUD: heterogeneous configuration optimization for distributed databases in the cloud, in: Gavrilovska, A., Zadok, E. (Eds.), 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020, USENIX Association. pp. 189–203.

[40] Miner, D., Shook, A., 2012. MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems. 1st ed., O'Reilly Media, Inc.

[41] Nykiel, T., Potamias, M., Mishra, C., Kollios, G., et al., 2010. MRShare: Sharing across multiple queries in MapReduce. Proc. of VLDB Endowment (PVLDB) 3, 494–505.

[42] Poggi, N., Berral, J.L., Fenech, T., Carrera, D., Blakeley, J., Minhas, U.F., Vujic, N., 2016. The state of SQL-on-Hadoop in the cloud, in: Proceedings of the 2016 IEEE International Conference on Big Data (Big Data), pp. 1432–1443.

[43] Presto, 2019. Presto: Distributed SQL Query Engine for Big Data. https://prestodb.github.io/. [Online; accessed 25-Feb-2021].

[44] Rajaraman, A., Ullman, J.D., 2011. Mining of Massive Datasets. Cambridge University Press.

[45] Sarma, A.D., Afrati, F.N., Salihoglu, S., Ullman, J.D., 2013. Upper and Lower Bounds on the Cost of a Map-Reduce Computation. Proc. of VLDB Endowment (PVLDB) 6.

[46] Shi, J., Zou, J., Lu, J., Cao, Z., Li, S., Wang, C., 2014. MRTuner: A Toolkit to Enable Holistic Optimization for MapReduce Jobs. Proc. of VLDB Endowment (PVLDB) 7, 1319–1330.

[47] Shvachko, K., Kuang, H., Radia, S., Chansler, R., et al., 2010. The Hadoop Distributed File System, in: Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10.

[48] Singhal, R., Singh, P., 2017. Performance Assurance Model for Applications on SPARK Platform, in: Proceedings of the Technology Conference on Performance Evaluation and Benchmarking (TPCTC), Springer. pp. 131–146.

[49] Song, G., Meng, Z., Huet, F., Magoules, F., Yu, L., Lin, X., 2013. A Hadoop MapReduce Performance Prediction Method, in: Proceedings of the IEEE 10th International Conference on High Performance Computing and Communications, pp. 820–825.

[50] Tajo, 2021. Tajo: A Big Data Warehouse System on Hadoop. https://tajo.apache.org/. [Online; accessed 25-Feb-2021].

[51] The Apache Software Foundation, 2020. Rumen: A tool to extract job characterization data form. https://hadoop.apache.org/docs/r1.2.1/rumen.html. [Online; accessed 25-Feb-2021].

[52] The Apache Software Foundation, 2021. SPIP: Support Stage level resource configuration and scheduling. https://issues.apache.org/jira/browse/SPARK-27495. [Online; accessed 25-Feb-2021].

[53] Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., et al., 2009. Hive: A warehousing solution over a Map-Reduce framework. Proc. of VLDB Endowment (PVLDB) 2, 1626–1629.

[54] TPC, 2021. TPC Benchmark H Standard Specification. http://tpc.org/TPC_Documents_Current_Versions/pdf/tpc-h_v2.18.0.pdf. [Online; accessed 25-Feb-2021].

[55] Van Aken, D., Pavlo, A., Gordon, G.J., Zhang, B., 2017. Automatic Database Management System Tuning Through Large-Scale Machine Learning, in: Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data, Association for Computing Machinery. p. 1009–1024. URL: https://doi.org/10.1145/3035918.3064029, doi:10.1145/3035918.3064029.

[56] Wang, G., Xu, J., He, B., 2016. A Novel Method for Tuning Configuration Parameters of Spark based on Machine Learning, in: Proc. of the 18th Intl. Conf. on High Performance Computing and Communications, IEEE. pp. 586–593.

[57] Wang, K., Lin, X., Tang, W., 2012. Predator — An experience guided configuration optimizer for Hadoop MapReduce, in: Proceedings of the 4th IEEE International Conference on Cloud Computing Technology and Science, pp. 419–426.

[58] Wu, D., Gokhale, A., 2013. A self-tuning system based on application Profiling and Performance Analysis for optimizing Hadoop MapReduce cluster configuration, in: Proceedings of the 20th Annual International Conference on High Performance Computing, pp. 89–98.

[59] Xin, R.S., Rosen, J., Zaharia, M., Franklin, M.J., Shenker, S., Stoica, I., 2013. Shark: SQL and rich analytics at scale, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of data, ACM. pp. 13–24.

[60] Yigitbasi, N., Willke, T.L., Liao, G., Epema, D., 2013. Towards machine learning-based auto-tuning of mapreduce, in: Proceedings of the 2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, pp. 11–20.

[61] Zhang, B., Křikava, F., Rouvoy, R., Seinturier, L., 2016. Self-balancing job parallelism and throughput in hadoop, in: Jelasity, M., Kalyvianaki, E. (Eds.), Proceedings of the Distributed Applications and Interoperable Systems, Springer. pp. 129–143.

[62] Zhang, Z., Cherkasova, L., Loo, B.T., 2013a. AutoTune: Optimizing Execution Concurrency and Resource Usage in MapReduce Workflows, in: Proceedings of the 10th International Conference on Autonomic Computing (ICAC), pp. 175–181.

[63] Zhang, Z., Cherkasova, L., Loo, B.T., 2013b. Benchmarking Approach for Designing a MapReduce Performance Model, in: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering, ACM. pp. 253–258. doi:10.1145/2479871.2479906.