Don't Tune Twice: Reusing Tuning Setups for SQL-on-Hadoop Queries

Edson Ramiro Lucas Filho[1], Eduardo Cunha de Almeida[1], and Stefanie Scherzinger[2]

[1] Universidade Federal do Paraná, Brazil
{erlfilho,eduardo}@inf.ufpr.br
[2] OTH Regensburg
stefanie.scherzinger@oth-regensburg.de

Abstract. SQL-on-Hadoop processing engines have become state-of-the-art in data lake analysis. However, the skills required to tune such systems are rare. This has inspired automated tuning advisers which profile the query workload and produce tuning setups for the low-level MapReduce jobs. Yet with highly dynamic query workloads, repeated re-tuning costs time and money in IaaS environments. In this paper, we focus on reducing the costs for up-front tuning. At the heart of our approach is the observation that a SQL query is compiled into a query plan of MapReduce jobs. While the plans differ from query to query, single jobs tend to be similar between queries. We introduce the notion of the *code signature* of a MapReduce job and, based on this, our concept of job similarity. We show that we can effectively *recycle* tuning setups from similar MapReduce jobs already profiled. In doing so, we can leverage any third-party tuning adviser for MapReduce engines. We are able to show that by recycling tuning setups, we can reduce the time spent on profiling by 50% in the TPC-H benchmark.

1 Introduction

More than a decade after the publication of the MapReduce paper [7], we observe a clear preference among Hadoop or Spark users for higher-level languages [11] (e.g., Hive [21] and SparkSQL [2]). Typically, writing queries for SQL-on-Hadoop systems is more productive than custom-coding MapReduce jobs for MapReduce frameworks: SQL-on-Hadoop systems compile declarative queries into a query plan of MapReduce jobs. Naturally, this greatly improves the productivity of data scientists. Yet compiling queries to query plans, and then allocating their jobs onto nodes in a cluster is only half the battle: The underlying MapReduce framework needs to be tuned for performance.

The expertise required for allocating the right mix of physical resources (main memory, disk space, bandwidth, etc.) to jobs, and for twiddling with the right tuning knobs is rare. This was already the case roughly 10 years ago, when the first automatic tuning advisers for MapReduce frameworks were proposed, e.g. [8]. Ever since, SQL-on-Hadoop engines and MapReduce frameworks have grown in complexity, manifesting in the number of tuning parameters. As Figure 1 shows, Hive [21] currently has about a thousand tuning parameters. Manual tuning is quite out of the question.

Tuning advisers for MapReduce frameworks rely on profiling of the query workload [12, 8, 1]. Naturally, profiling imposes an overhead. For instance, the Starfish tuning adviser causes an overhead of up to 50% [12]. When the query workload is highly dynamic, re-tuning becomes a cost factor in pay-as-you-go IaaS environments.

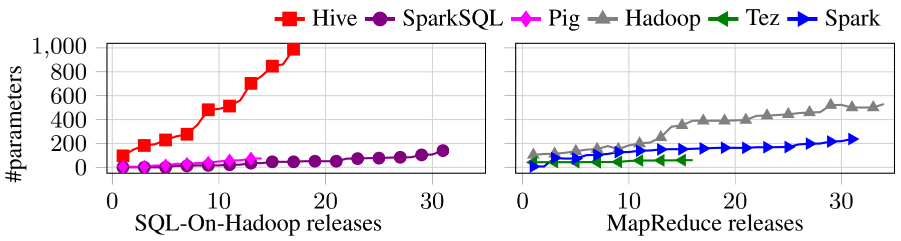


Fig. 1: Number of tuning parameters growing over time [10].[3]

In this paper, we model the static information of Hive to match jobs with similar resource consumption patterns and reuse the same tuning setup to reduce their total cost of tuning. Our model relies on two observations regarding the query plans compiled from SQL queries: (a) The jobs within a query plan execute different query operators, and often have different resource requirements. (b) Since the jobs are generated automatically, MapReduce jobs tend to be similar *across* query plans.

Let us consider a specific example regarding observation (a). Figure 2 shows the resource consumption of TPC-H query 5. The Hive engine (version 0.6.0) compiles this query into a sequence of seven MapReduce jobs. For each job, we track main memory and CPU consumption, as well as the amount of data in physical reads and writes. Evidently, the individual jobs have different *profiles* w.r.t. to their resource consumption.

Let us now consider observation (b). We regard two MapReduce jobs in two query plans as similar from the perspective of tuning, if they have the same *code signature*. Intuitively, the code signature of a MapReduce job captures the SQL operators implemented by this job, as well as the expected size of the input. This information is available through the Hive query compiler. Our hypothesis (which we can confirm in our experiments) is that jobs that share the same code signature benefit from the same tuning setups. We therefore *reuse* tuning setups for similar jobs to reduce profiling time.

Let us illustrate this point. Compiling the TPC-H queries in Hive-0.6 yields 123 MapReduce jobs. For 75% of these jobs, there is at least one other job with the same code signature. Only a quarter of all jobs has a unique code signature.

In fact, once we have profiled enough jobs, we may even be able to assign tuning setups for ad-hoc queries. These are queries that we have not encountered (or profiled) yet. In fact, ad-hoc queries are prevalent in many query workloads [23], yet tuning advisers for MapReduce frameworks rely on profiling the workload up front. Thus, our approach can be used in environments where, traditionally, tuning advisers fail.

In Section 2, we review the state-of-the-art on SQL-on-Hadoop engines, as well as MapReduce frameworks and their tuning advisers. In Section 3, we motivate and define the notion of the code signature of a MapReduce job, and introduce the code signature cache. We conduct our experiments using the TPC-H queries in Section 4. In Section 5, we discuss related work in the context of our approach. We conclude with Section 6.

[3] The releases are: Hadoop from 0.23.11 to 2.8.0, Spark from 0.5.0 to 2.2.0, Tez from 0.5.0 to 0.8.5, Hive from 0.3.0 to 2.3.0, Pig from 0.1.0 to 0.16.0, and SparkSQL from 1.1.0 to 2.2.0.

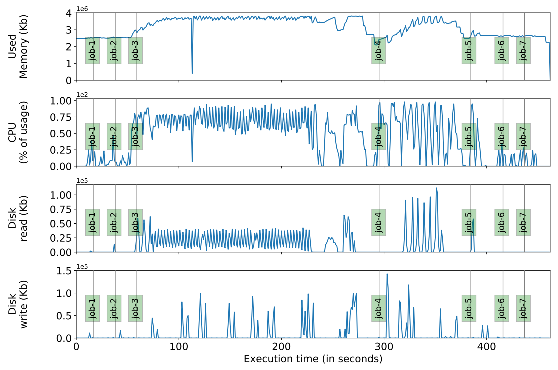


Fig. 2: CPU, memory, disk write and disk read consumption for TPC-H query 5 across the 7 MapReduce jobs.[4]

2 State-of-the-Art

We briefly sketch the control flow of compiling queries in SQL-on-Hadoop engines. We then describe how automatic tuning advisers for MapReduce frameworks proceed.

2.1 SQL-on-Hadoop Engines

The generic workflow within a SQL-on-Hadoop engine starts when a SQL query is submitted to the *Driver*. This component manages session handlers and tracks statistics. The *Compiler* then translates the query into a logical query plan. The *Optimizer* rewrites the logical plan in order to find a good execution plan in terms of execution costs. For instance, joins sharing the same join predicate may be merged, or data partitions irrelevant to query evaluation may be disregarded. The *Executor* then receives the DAG of MapReduce jobs. It queues the jobs in the MapReduce framework for processing. The MapReduce jobs are nodes in a directed acyclic graph (DAG). The directed edges denote dependencies between jobs.

Example 1. Figure 3b shows the final query plan produced by Hive v0.6.0 for TPC-H query 5 (presented in Figure 3a). Each job is responsible for executing one or more SQL operators in the query, like sort and aggregation. □

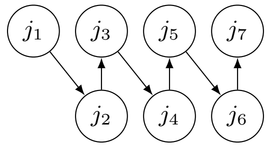
The DAG declares a partial order, which the *Execution Engine* considers when deploying the jobs. However, the MapReduce framework needs to be configured for performance: A *tuning setup* is registered with the MapReduce framework before the jobs can be executed. Today’s SQL-on-Hadoop engines assign a single tuning setup to all jobs of a query. However, technically, the underlying MapReduce framework allows each job to run with its own tuning setup.

Tuning advisers for MapReduce frameworks, on the other hand, have not been designed to tune a DAG of jobs, such as a query plan compiled from SQL queries. Rather, they produce one tuning setup per MapReduce job, as discussed next.

[4] We executed the experiments on a cluster of 3 nodes (Intel(R) Core(TM) i3-3240 CPU @ 3.40GHz, 4 GB of RAM and 1 TB of disk space each). Details are provided in Section 4.

```
insert overwrite table q5_local_supplier_volume
select n_name, sum(l_extendedprice * (1 − l_discount)) as revenue
from customer c join
    ( select n_name, l_extendedprice, l_discount, s_nationkey, o_custkey
      from orders o join
      ( select n_name, l_extendedprice, l_discount, l_orderkey, s_nationkey
        from lineitem l join
        ( select n_name, s_suppkey, s_nationkey from supplier s join
          ( select n_name, n_nationkey from nation n join region r
            on n.n_regionkey = r.r_regionkey and r.r_name = 'ASIA'
          ) n1 on s.s_nationkey = n1.n_nationkey
        ) s1 on l.l_suppkey = s1.s_suppkey
      ) l1 on l1.l_orderkey = o.o_orderkey
      and o.o_orderdate >= '1994−01−01' and o.o_orderdate < '1995−01−01'
) o1 on c.c_nationkey = o1.s_nationkey and c.c_custkey = o1.o_custkey
group by n_name order by revenue desc;
```

(a) TPC-H query 5.



(b) Query plan.

Fig. 3: Query and query plan of TPC-H query 5, as compiled by Hive 0.6.0.

2.2 Tuning Advisers for MapReduce Frameworks

In general, there are different strategies for obtaining a profile of the resource requirements of a MapReduce job. For instance, tuning advisers consider MapReduce Job-Counters [16, 15], real-time statistics [22], job execution time [3] or phases execution time [4], instrumentation of the JVM [17, 12], or perform log analysis [19, 20].

Inevitably, profiling adds an overhead to the execution time of a MapReduce job. For instance, Starfish [12] instruments the JVM. When Starfish monitors all of the JVM tasks, the authors report a profiling overhead of 50%. To speed up tuning, Starfish can be configured to profile only a sample of the JVM tasks. For instance, when profiling only 20% of the JVM tasks, the profiling overhead drops to 10%. However, not sampling but profiling all tasks will lead to more effective tuning setups.

Reusing job profiles is a way to reduce the cost of tuning. The authors of PStorM [9] propose a form of sampling to reduce the tuning overhead. They sample only a single map and reduce task. This produces a *tiny-profile*, which they match against a history of profiles. After PStorM finds a match to its tiny-profile, it feeds the Tuning Adviser (e.g., Starfish) with the match in order to generate a tuning setup.

Internally, tuning advisers maintain a representation of the tuning parameters and their domains, as well as a cost model to predict the execution time of a given job. They also apply heuristics. It is beyond the scope of this paper to do a more thorough survey, especially as we employ tuning advisers as black box systems. However, in order to give an idea of both the complexity of the problem, as well as the plethora of solutions proposed, we list prominent tuning advisers in Table 1. The reported speedups range from 24% up to 7.4x, depending on the workload. Note the richness of heuristics applied in the various tools.

3 The Code Signature Cache

At the heart of our approach is a data structure called the *code signature cache*. It assigns tuning setups to MapReduce jobs. The tuning setups are produced by a third-

Tuning System	Speed up	Hadoop Version	Heuristics
MR-COF [17]	up to 41%	0.20.2	Genetic Algorithm
Gunther [16]	up to 33%	0.20.3	Genetic Algorithm
Starfish [12]	up to 46%	0.20.2	Random Recursive Search
MRTuner [19]	1x	1.0.3 and 1.1.1	PTC-Search
Panacea [18]	1.6x up to 2.9x	-	Exhaustive Search
RFHOC [4]	2.11x-7.4x	1.0.4	Genetic Algorithm
GeneExpression [14]	46%-71%	1.2.1	Particle Swarm Optimization
MROnline [15]	30%	2.1.0	Hill Climbing
MEST [3]	-	2.6.0	Genetic Algorithm
JellyFish [22]	up to 74%	YARN	Hill Climbing

Table 1: Tuning advisers for MapReduce frameworks in comparison: Reported speedup, supported Hadoop version, and the heuristics used to determine tuning setups.

party tuning adviser. We use the *code signature* of a MapReduce job as lookup key in the code signature cache. We introduce the notion of a code signature shortly. We then discuss how the code signature cache manages tuning setups.

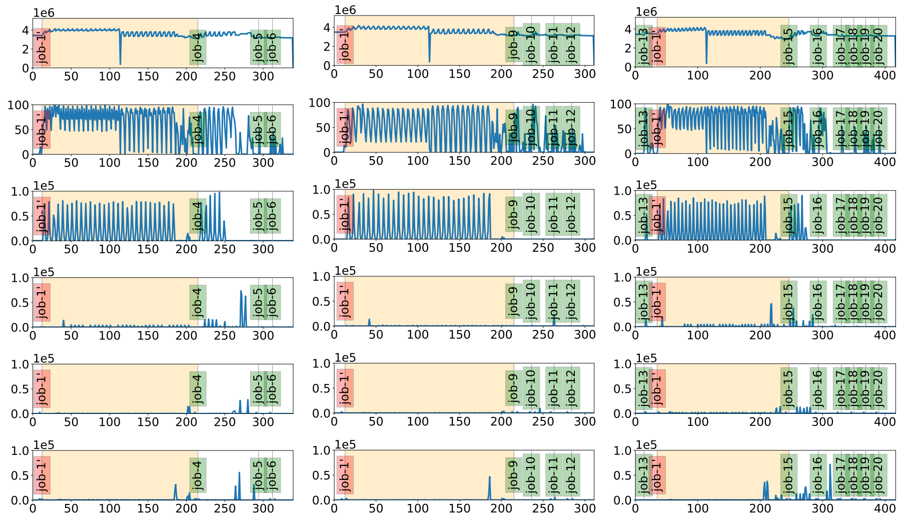
3.1 Code Signatures

The SQL-on-Hadoop engine Hive compiles SQL queries to query plans. During query compilation, the resulting MapReduce jobs are annotated with several descriptive properties. The Hive Java API makes these annotations accessible: Each job is annotated with a list of the physical query operators that are implemented by this job. For each operator, a *cardinality* is given. For instance, a job may execute two *Filter*-operations (i.e. selection in relational algebra), as well as one aggregation. Each job is further annotated with the estimated size of its input.

Example 2. TCP-H query 1 is compiled by Hive into two jobs. The first job is annotated with the operators Filter, Select, and GroupBy. Each operator has cardinality 2. The job is further annotated with the operators TableScan, ReduceSink, and FileSink, each with a cardinality of 1. In the setup of our experiments (see Section 4) the estimated input size is stated as 7.24GB. □

Our assumption is that these declarative annotations are related to the resource profile of these jobs. Further, we hypothesize that jobs with the same annotations may be executed with the same tuning setups, even though they differ in their Java code. In our experiments, we are able to confirm this. For now, we argue on an intuitive level.

Example 3. Let us consider Figure 4, where we have run selected TPC-H queries with a default tuning setup. For each query, we execute the MapReduce jobs according to the query plan. Evidently, the queries have different query plans, and therefore a different number of jobs. However, visual inspection suggests that certain jobs have similar resource profiles. Thus, we have highlighted these jobs with a shaded background. Incidentally, these jobs also share the same code signature. In Figure 4, we have labeled jobs with a unique code signature with a unique job identifier (e.g., jobs 2, 3, and 4), and we have labeled the jobs with the same code signature as job 1'.



(a) TPC-H Query 4 (b) TPC-H Query 15 (c) TPC-H Query 20

Fig. 4: Execution of selected TPC-H queries. The jobs labeled 1' share the same code signature and apparently have similar resource profiles. This suggests that they would benefit from the same tuning setup.

Thus, this suggests that tuning setups may be shared between jobs with the same code signature: Once one instance of job 1' has been profiled, we reuse its tuning setup for the other jobs with the same signature. Thus, we simply skip profiling these jobs. □

In fact, similarity of resource consumption between common MapReduce jobs (e.g., sort, grep, WordCount) has already been identified [13]. However, we believe to be the first wort to model Hive jobs specifically to identify this similarity. The main difference of our model is that it calculates the code-signature of a given job at compiling time, instead of running or sampling it.

Since the MapReduce jobs are compiled from queries, the query plans of syntactically different queries nevertheless often contain jobs with the same query annotations. Formally, we capture these annotations by the *code signature* of a job, as defined next.

3.2 Definitions

We now embark on the formal definitions. Let us define a *query plan* as a directed acyclic graph $G = (V, E)$, where the set of vertices V represents the MapReduce jobs, and the set of edges E denotes the precedence between two jobs. More precisely, a vertex (job) $j \in V$ is a tuple of the form $v = (O_j, T_j, C_j)$ in which O_j is the set of physical *query operators* it executes (the set of physical query operators is fixed, $O_j = \{o_1, \ldots, o_n\}$. For instance, Hive version 0.6.0 knows 16 different physical operators), T_j is the set of *associated input tables*, and C_j is the set of *configurations* used to allocate resources. Each directed edge $e \in E$ is an ordered pair of vertices defined as $e = (i, j)$ and connects the jobs i to j, when the execution of i directly precedes j.

The query compiler assigns the implemented physical query operators, as well as their cardinalities, to each node in the query plan. We thus consider the annotation function $ops : V \times O \rightarrow \mathbb{N}_0^+$, where

$$ops(j, o) = \begin{cases} n \text{ job } j \text{ implements operator } o \text{ exactly } n \text{ times} \\ 0 \text{ otherwise.} \end{cases}$$

We also consider the annotation function $ord : V \rightarrow \mathbb{N}_0^+$, which returns the order of magnitude of the expected input data for the given MapReduce job.

We are now in the position to define the code signature of a MapReduce job.

Definition 1. The *code signature* of a MapReduce job j in V is a $(|O| + 1)$-tuple,

$$\textit{codesignature}(j) = \langle ord(j),\ o_1 : c_1,\ \ldots,\ o_i : c_i,\ \ldots,\ o_n : c_n \rangle$$

where $c_i = ops(j, o_i)$, the cardinality of this operator. □

In the following example, we omit operators from the code signature with a cardinality of zero, for the sake of brevity.

Example 4. We continue with TPC-H query 1. The code signature of the first job is

$$\langle 9,\ \textit{Tablescan} : 1,\ \textit{Filter} : 2,\ \textit{Select} : 2,\ \textit{Groupby} : 2,\ \textit{Reducesink} : 1,\ \textit{Filesink} : 1 \rangle$$

The order of magnitude of the input size is 9. □

3.3 Cache Hits and Misses

Figure 5 visualizes the code signature cache. Initially, the cache is empty. A SQL query is compiled by Hive into the query plan (Step 1). For each of the jobs $j_1, \ldots, j_n$ in this query plan, we look up the tuning setup in the code signature cache (Step 2). For each cache miss, we employ the Starfish tuning adviser for profiling the job and generating a tuning setup (Step 3). The tuning setups, denoted $t_1, \ldots, t_n$, are stored in the code signature cache, with the code signatures of the jobs as lookup keys (Step 4). As the cache becomes populated, we observe more cache hits (Steps 5 and 6). In the best case, we have cache hits for all jobs in the query plan. Then we can simply reuse the tuning setups of similar jobs, and need not turn to Starfish for profiling at all.

4 Experiments

We have implemented the code signature cache in Java, and integrated it with Apache Hive. We leverage tuning setups generated by the Starfish tuning adviser [12]. Unless explicitly stated otherwise, we run Starfish with sampling turned off (i.e., a sampling rate of 100%), to obtain high-quality tuning profiles. Using Starfish 0.3.0[5], we are tied to Hadoop 0.20.2 and Hive 0.6.0. We point out that the code signature cache is a generic data structure and not restricted to any particular version of Hive or Hadoop.

[5] The Starfish binary is available at https://www.cs.duke.edu/starfish/release.html.

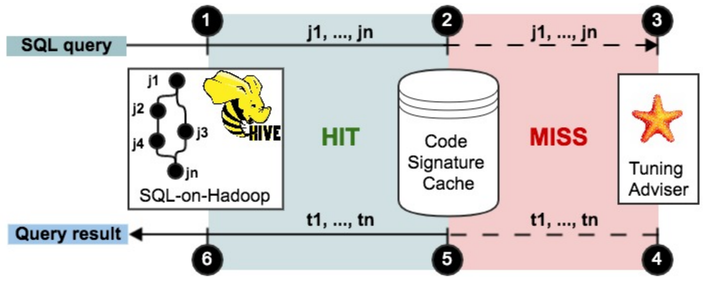


Fig. 5: Lookups in the code signature cache. For a cache miss, a third-party tuning adviser is run to produce a tuning setup.

We evaluate the TPC-H queries provided for Hive[6]. The data has been generated with a scale factor of 10. This amounts to 10.46GB of data when stored on disk.

Our experiments were executed in a cluster with three physical machines. We isolate the master node on one machine, so that it does not influence with the profiling of jobs.

In particular, each machine has a Intel(R) Core(TM) i3-3240 CPU @ 3.40GHz, 4GB of RAM, 1TB of disk. We used the *collectl* tool[7] tool to measure CPU, memory, network, and disk consumption. The reported execution times are averaged over 10 runs. All our profiling runs are configured with the out-of-the-box tuning setup that we refer to as ”Hadoop Standard”. We first consider the reuse of tuning setups at the level of single MapReduce jobs, and later at the level of SQL queries.

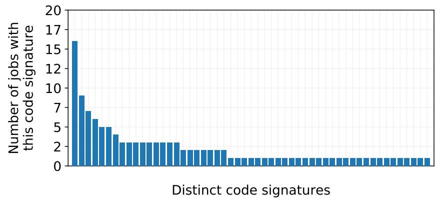
4.1 Recycling Tuning Setups at the Job Level

We first study the distribution of code signatures in the query plans of TPC-H queries. We then confirm that the code signature is indeed a viable basis for recycling tuning setups among jobs.

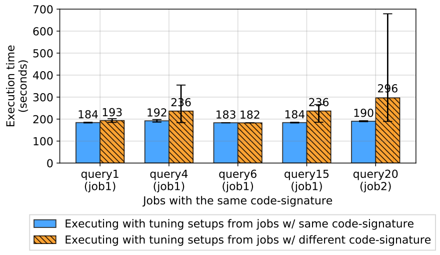
Repeating code signatures We have compiled the TPC-H queries into query plans. Figure 6a shows the number of jobs with the same code signature. There is one code signature that is actually shared by 16 jobs. In fact, this job occurs in over 70% of all TPC-H queries. Moreover, for 75% of all MapReduce jobs, there is at least one other job with the same code signature. Only a quarter of all jobs has a unique code signature. Thus, there is a considerable share of recurring code signatures.

Justifying the recycling of tuning setups We experimentally examine our hypothesis, stating that we may recycle tuning setups for jobs with the same code signature in Figure 6b. We choose 5 representative MapReduce jobs that all share the same code signature. When compiled into query plans, we obtain 20 MapReduce jobs. For each job j of those five jobs, we define two groups of jobs:

[6] See https://issues.apache.org/jira/browse/HIVE-600 for the verbatim SQL queries.
[7] http://collectl.sourceforge.net



(a) MapReduce jobs compiled from TPC-H queries: Counting jobs that share the same code signature.



(b) Jobs with the same code signature tend to benefit from the same tuning setups. This effect cannot be repeated for jobs with different code signatures.

Fig. 6: Recycling tuning setups among jobs with the same code signature.

1. The five jobs that share the same code signature. For these jobs, we obtain the 5 tuning setups from Starfish.
2. The remaining jobs within the same query plan. These have different code signatures. Again, we obtain the tuning setups from Starfish.

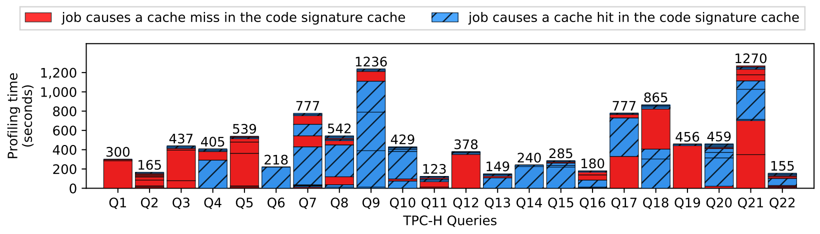
We then execute job 1 of query 1 with all the tuning setups from groups (1) and (2), shown in the first and second bar respectively. We repeat this procedure for the other jobs listed. In general, the jobs executing with the tuning setups of group (1) show better performance than with the tuning setups of group (2). The reported execution times are averaged over 10 runs. The error bars mark the minimum and maximum execution times. There is noticeably less variance in the execution times of group (1).

Overall, we see that for jobs with the same code signature, we may use the tuning setups interchangeably. When jobs have different code signatures, this is not necessarily the case. We have conducted this experiment for all recurring code signatures, and we have made the same observations in the other cases as well. For the sake of conciseness, we show only the case portrayed in Figure 6b.

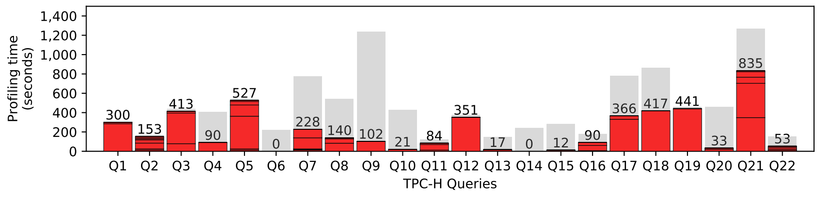
4.2 Recycling Tuning Setups at the Query Level

We now employ the code signature cache for profiling the TPC-H queries. We first profile all 22 queries in the order specified by the benchmark and discuss the benefits of applying the code signature cache. We then contrast this with the total time spent on profiling if Starfish only samples the JVM tasks. We also consider different execution orders in profiling the TPC-H queries with the code signature cache, and show that the query order does not have as much impact on profiling time as one might expect. Finally, we compare the execution time of non-uniform tuning when we have the code signature cache available during profiling and when we profile all jobs with Starfish.

Profiling the TPC-H queries in order We profile the 22 TPC-H queries in the order of the TPC-H benchmark specification. Figure 7a shows the profiling time per query. In total, over ten thousand seconds are spent on profiling.



(a) Total time Starfish spends profiling (no sampling): 10,396.97 seconds.



(b) Reduced profiling time leveraging the code signature cache. Total time spent profiling: 4,679.66 seconds.

Fig. 7: The code signature cache reduces the profiling time by over 50% for TPC-H.

Even though the runs in Figure 7a do not make use the code signature cache, for the purpose of illustration, we visually distinguish two groups of jobs:

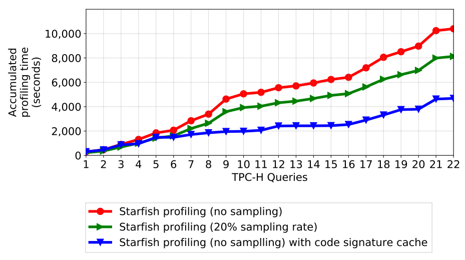
1. Jobs which cause a cache miss in the code signature cache,
2. and jobs which cause a cache hit in the code signature cache.

We can observe that for the first TPC-H query, all jobs would cause a cache miss. Yet already for the second and third queries, we'd have cache hits, even though the savings are minor. With the code signature cache becoming more populated, we get more cache hits, and in some cases, some substantial savings in the profiling time. For instance, for queries Q6 and Q14, we can recycle all tuning setups from the cache. Thus, they require no profiling at all.

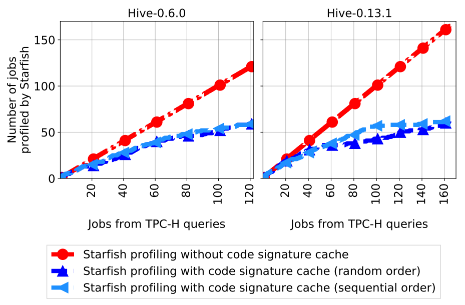
Let us now turn to the quantitative assessment. In Figure 7b, we use the code signature cache. Thus, we employ Starfish only for the jobs from group (1), and recycle the tuning setups for the jobs from group (2). The shaded grey area indicates the height of the original bars from Figure 7a, for easier comparison.

Using the code signature cache reduces the total time spent on profiling from over ten to below five thousand seconds. Overall, we can cut down the time spent profiling by more than 50%.

Recycling vs. Starfish sampling Starfish has its own strategy for reducing the profiling time, by sampling only a share of the JVM tasks. We compare this strategy with our approach of using Starfish (without sampling) in combination with the code signature cache. In Figure 8a, we compare the total accumulated time spent on profiling for different modes of operation. The topmost line denotes profiling with Starfish, where sampling is turned off. This summarizes the experiment of Figure 7a.



(a) Accumulated time for profiling all queries.



(b) Varying the order in query profiling.

Fig. 8: Accumulated time for profiling.

When we run Starfish with a sampling rate of 20% (nevertheless executing all tasks of the query), the total time spent on profiling is effectively reduced. However, sampling increases the error rate in the resulting tuning profiles [12].

In the given chart, the profiling time is lowest for the combination of Starfish and recycling from the code signature cache. Thus, we can profile in half of the time, without having to make the sacrifices due to sampling.

Varying the query order The recycling rate of tuning setups in the code signature cache, as reported in Figure 7b, is influenced by the order in which the TPC-H queries are profiled.

In Figure 8b, we vary the order of queries. Moreover, we contrast the MapReduce jobs produced by two different versions of Hive. On the horizontal axis, the charts show the total number of MapReduce jobs compiled from the TPC-H queries by the Hive query compiler. On the vertical axis, we denote the number of jobs which had to be profiled by Starfish.

The tuning adviser Starfish, when used stand-alone, profiles all jobs. We further compare Starfish in combination with the code signature cache. Regardless whether the queries are encountered in order of their specification or in a randomly generated order, for this query workload, we can recycle tuning setups from the cache for about half of the jobs.

Thus, while the query compilers of Hive 0.6.0 and 0.13.1 produce a different number of jobs, the benefits of recycling is independently of the submission order.

Comparing execution times Finally, we execute the TPC-H queries with non-uniform tuning. In one scenario, the tuning setups for all jobs have been generated by Starfish. In a second scenario, we have recycled tuning setups from the code signature cache.

In total, this yields a speedup (or rather, slowdown) of 0.93 for the TPC-H queries. Thus, thanks to the code signature cache, we only spend about half the profiling time, with a tolerable impact on query execution times.

This is a good result, considering that Starfish with sampling turned on imposes a higher error rate (e.g., an error rate of 15% when sampling merely 10% of the JVM tasks [12]).

4.3 Discussion

In summary, we can experimentally support our hypothesis that jobs with the same code signatures benefit from the same tuning setups. Therefore, we may recycle tuning setups. By reducing the number of jobs to be profiled, we can effectively cut down on the time required for physical-level performance tuning.

Even when Starfish profiles only a sample of 20% of the tasks in the JVM, it does not reach this speedup (while the quality of tuning setups produced by Starfish degrades). Thus, coupling a third-party tuning advisor with the code signature cache is a winning strategy for reducing profiling time.

For some queries, we were even able to directly assign tuning setups to MapReduce jobs, requiring no profiling at all. This is promising for processing ad-hoc queries, which normally do not benefit from up-front tuning. In our experiments with the TPC-H queries, we were able to cut down profiling time by half. Moreover, the mechanism is quite robust when the order of TPC-H queries varies.

5 Related Work

Performance tuning for database management systems is an evergreen in database research, and a profitable consultancy business in industry. As stated by Bonnet and Shasha [5]: "An under-appreciated tuning principle asserts start-up costs are high; running costs are low."

There are several projects aiming at reducing the profiling time in physical-level tuning. In Section 2, we have already described how Starfish [12] samples the JVM tasks executing MapReduce jobs. This reduces the profiling time, but at the cost of tuning effectiveness.

The PStorM [9] system is closest to our work. As in our approach, PStorM leverages Starfish as a third-party tuning adviser. Also similar to our idea of caching tuning setups, similar jobs are mapped to existing tuning setups in a profile store.

However, our notion of job similarity based on code signatures is much simpler, since we rely on the declarative annotations that the Hive query compiler adds to MapReduce jobs. In contrast, PStorM considers code similarity metrics, and compares control flow graphs (CFG) as well as feature vectors. PStorM avoids computing a code signature (e.g., by hashing the source code or byte code), because of the risk of mismatching source code with similar behaviour, but different code primitives (e.g., for-loop vs. while-loop).

In particular, PStorM analyzes the byte code for static features and samples map tasks for dynamic features to probe the profile store for matching profiles. Thus, the matching accuracy depends on the size of the sample and the maintenance of the features, which also incur an execution overhead. After all, creating, maintaining and testing feature vectors is time consuming.

Our code signature cache is less complex, yet nevertheless highly effective: The lookup key is based on declarative query operators. Thus, we operate on a higher level of abstraction, instead of on the underlying code primitives.

Another related system is Kambatla [13], that monitors the execution of a given job using a pre-defined number of intervals. It computes the average consumption for each resource within each interval and matches it with similar profiles. MrEtalon [6] is another related system that profiles a given job on a sample of the data set. MrEtalon builds a similarity matrix to this profile and compares it to the pre-established similarity matrices, that will generate the recommended tuning setup. All in all, the drawback of these systems is that they increase the time spent on the tuning activity due to the effort to sample the number of MapReduce tasks or the data set to match similar jobs.

6 Conclusion

Automated tuning of SQL-on-Hadoop engines and MapReduce frameworks is a highly topical research area. Tuning adviser tools profile MapReduce jobs to produce suitable tuning setups. Naturally, profiling introduces an overhead. In pay-as-you-go environments, profiling can drive up the operational costs considerably. Therefore, ways for reducing the time spent on tuning are of great interest to the research community and practitioners alike.

Existing tuning advisers cut down the profiling time by sampling, either monitoring only a share of the JVM tasks (as done by Starfish), by monitoring only a specific sample of MapReduce jobs (as done by PStorM), or running the jobs with a sample of its data set. Thus, they trade time for the effectiveness of the resulting tuning setup.

In this paper, we reduce the profiling time by skipping profiling altogether for MapReduce jobs where we can recycle the tuning setups from similar jobs. To this end, we rely on our model of the code signature as a means for identifying similar jobs, and to populate the code signature cache.

Our approach is appealingly simple, yet effective, and lets us cut back on profiling by nearly 50% in case of the TPC-H queries, without major sacrifices to the quality of tuning setups. Provided that we have successfully profiled enough similar queries (or rather, their MapReduce jobs), we may even supply ad-hoc queries with tuning setups, skipping up-front profiling altogether.

As future work, we plan to refine the code signature cache by integrating the selectivity of query operators into the code signature. Moreover, we hope to be able to use the code signature cache for application-level tuning as well: By caching power hints for MapReduce jobs, we might be able to automatically suggest performance hints for similar jobs. This could be a great relief to the data analyst who has no prior background in database administration.

Acknowledgments

We thank Herodotos Herodotou for all the support with Starfish. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

References

1. Aken, D.V., Pavlo, A., Gordon, G.J.: Automatic Database Management System Tuning Through Large-scale Machine Learning. In: SIGMOD (2017)
2. Armbrust, M., Ghodsi, A., Zaharia, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J.: Spark SQL: Relational Data Processing in Spark. In: SIGMOD (2015)
3. Bei, Z., Yu, Z., Liu, Q., Xu, C., Feng, S., Song, S.: MEST: A Model-Driven Efficient Searching Approach for MapReduce Self-Tuning. IEEE Access (2017)
4. Bei, Z., Yu, Z., Zhang, H., Xiong, W., Xu, C., Eeckhout, L., Feng, S.: RFHOC: A Random-Forest Approach to Auto-Tuning Hadoop's Configuration. IEEE Transactions on Parallel and Distributed Systems **27**(5), 1470–1483 (2016)
5. Bonnet, P., Shasha, D.E.: Application-level tuning. In: Encyclopedia of Database Systems (2009)
6. Cai, L., Qi, Y., Li, J.: A Recommendation-Based Parameter Tuning Approach for Hadoop. In: International Symposium on Cloud and Service Computing, SC2 2017 (2018)
7. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI (2004)
8. Duan, S., Thummala, V., Babu, S.: Tuning Database Configuration Parameters with iTuned. ReCALL **2**(1), 1246–1257 (aug 2009)
9. Ead, M.: PStorM: Profile Storage and Matching for Feedback-Based Tuning of MapReduce Jobs. In: EDBT (2014)
10. Filho, E.R.L., de Melo, R.S., de Almeida, E.C.: A Non-Uniform Tuning Method for SQL-on-Hadoop Systems. In: AMW (2019)
11. Floratou, A., Minhas, U.F., Özcan, F.: SQL-on-Hadoop: Full Circle Back to Shared-Nothing Database Architectures. In: PVLDB. vol. 7, pp. 1295–1306 (2014)
12. Herodotou, H., Lim, H., Luo, G., Borisov, N., Dong, L., Cetin, F.B., Babu, S.: Starfish: A Self-Tuning System for Big Data Analytics. In: CIDR (2011)
13. Kambatla, K., Pathak, A., Pucha, H.: Towards Optimizing Hadoop Provisioning in the Cloud. Design (2009)
14. Khan, M., Huang, Z., Li, M., Taylor, G.A., Khan, M.: Optimizing hadoop parameter settings with gene expression programming guided PSO. Concurrency Computation (2017)
15. Li, M., Zeng, L., Meng, S., Tan, J., Zhang, L., Butt, A.R., Fuller, N.: MRONLINE: mapreduce online performance tuning. In: HPDC (2014)
16. Liao, G., Datta, K., Willke, T.L., Kalavri, V., Vlassov, V., Brand, P.: Gunther: Search-Based Auto-Tuning of MapReduce. Euro-Par (2013)
17. Liu, C., Zeng, D., Yao, H., et al: MR-COF: A Genetic MapReduce Configuration Optimization Framework. In: Theoretical Computer Science (2015)
18. Liu, J., Ravi, N., Chakradhar, S., Kandemir, M.: Panacea: Towards Holistic Optimization of MapReduce Applications. In: CHO (2012)
19. Shi, J., Zou, J., Lu, J., Cao, Z., Li, S., Wang, C.: MRTuner: A Toolkit to Enable Holistic Optimization for MapReduce Jobs. In: PVLDB. vol. 7, pp. 1319–1330 (2014)
20. The Apache Software Fundation: Rumen: A Tool to Extract Job Characterization Data form Job Tracker Logs. (2013), https://hadoop.apache.org/docs/r1.2.1/rumen.html
21. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Anthony, S., Liu, H., Murthy, R.: Hive - a petabyte scale data warehouse using hadoop. In: ICDE (2010)
22. Xiaoan Ding, Yi Liu, Depei Qian, et al: JellyFish: Online Performance Tuning with Adaptive Configuration and Elastic Container in Hadoop YARN. In: ICPADS (2016)
23. Yanpei Chen, S.A., Katz, R.H., Chen, Y., Alspaugh, S., Katz, R.: Interactive Query Processing in Big Data Systems: A Cross Industry Study of MapReduce Workloads. Tech. Rep. 12, University of California, Berkeley (aug 2012)