Towards Selecting Best Combination of SQL-on-Hadoop Systems and JVMs

Tatsuhiro Chiba, Takeshi Yoshimura, Michihiro Horie and Hiroshi Horii
IBM Research - Tokyo
19-21, Nihonbashi Hakozaki-cho
Chuo-ku, Tokyo, 103-8510, Japan
Email: {chiba, tyos, horie, horii}@jp.ibm.com

***Abstract*—While Hadoop is the de facto standard big-data middleware, many frameworks have been developed on top of it. Since many SQL-on-Hadoop systems are available, we often consider which engine is best for our queries. We can choose not only query engines but also Java virtual machines (JVMs) as well. As their systems become more complex, however, it is not always true that a single system performs best at any time. Moreover, the performance of a mismatched system may degrade greatly. To exploit the best performance, it is important to know what type of queries are suitable for a system and then to schedule queries for the appropriate system. In this paper, we evaluated the TPC-DS benchmark on a combination of query engines (Spark and Tez) and JVMs (J9 and OpenJDK). We found that using different engines lead to a drawback of over 10 times and that using different JVMs leads to a drawback of 3 times. We also analyzed the characteristics of each combination and then proposed classification models for selecting the best combination of systems with a generated query plan. As a result, we achieved a performance improvement of up to two times in total with the classifier.**

I. Introduction

As data volumes have increased, large-scale distributed data processing frameworks have been developed and widely used in many applications. Hadoop [1] is the most popular open-source framework for enabling scale-out data processing with fault tolerance. Consequently, many data-centric applications, such as extract-transform-load (ETL), relational query processing, and machine learning, have been recently built on top of the Hadoop eco-system. Although various frameworks have been proposed for effectively processing massive amounts of data in Hadoop, relational processing on Hadoop, i.e., SQL-on-Hadooop, still remains a hot topic of user interest [2]. Thus, many SQL-on-Hadoop systems, such as Hive [3], Spark SQL [4], Impara [5] and Presto [6], have been developed.

Users often consider which system is suitable for their queries on the basis of a performance comparison or research papers [7], [8]. Once they have decided which system to select from among the candidates, they may never consider whether another candidate system could perform better with newly integrated features. There are several reasons users do not move to other systems. First, system migration is very time-consuming. In addition to the fact that extensive expert knowledge on the characteristics and the details of system implementation is sometimes required to achieve the optimal performance, users need to take into consideration the details of a new system again and re-implement a suitable SQL. Second, the complexity of systems is growing. Even when a simple query is being performed, computation depends on many libraries, frameworks, runtimes, the operating system, and so on. Moreover, a fair head-to-head comparison is quite difficult since surrounding libraries and assumptions differ. Hence, this diversity of systems often locks us into a single system.

Meanwhile, many frameworks on top of Hadoop are written in Java or Scala; therefore, their performance significantly depends on runtime, i.e., Java virtual machines (JVMs). Since JVMs have also been developed in open-source communities, we can choose a suitable JVM from multiple implementations such as OpenJDK and J9. As JVM specifications are strictly defined, we can run any Java bytecode on a selected JVM without there being a difference in execution result. However, each JVM has differently implemented core components. For example, the garbage collection (GC) algorithm, heuristics of the just-in-time (JIT) compiler, and internal memory layout of JVMs often differ from each other. Consequently, the performance characteristics also differ depending on the application as well as query engine. Since performance bottlenecks have been moving to the CPU and memory layer due to the recent trend in in-memory processing, the differences in JVMs influence application performance more than ever before. Therefore, it is also important to consider these differences and select an adequate JVM to achieve better performance by exploiting the advantages.

To exploit the advantages of underlying middleware, i.e., query engines and JVMs, in this paper, we argue that query execution should be decoupled from systems once and an adequate coupling of systems should be conducted again at runtime. This interoperability would help to greatly improve not only query performance but also resource utilization, e.g., a cloud computing model that decouples physical resources from logical services. One of the advantages of a system that performs in the Hadoop eco-system is that data can be shared over the Hadoop Distributed File System (HDFS). Without data modification, users can get query results as quickly as possible by scheduling queries on adequate systems. At the same time, minimizing processing time introduces great benefits regarding higher resource utilization on the whole.

There are several challenges with scheduling a query for the

best suitable system. First, we need to characterize the target system with a detailed analysis. Identifying the performance trade-offs in query engines and JVMs is mandatory for taking full advantage of every system. Second is making a decision model on the basis of machine learning. If the execution environment changes, the decision model may change as well. Thus, the change in model should rely on the observation of actual system performance. Third is updating the model periodically when a new version of a system is available. Rebuilding a model from scratch is sometimes very time consuming when a new version is deployed on a system, so it is important to update the model without breaking the current model as much as possible.

In this paper, we addressed the first two challenges: system characterization and machine-learning-based decision model. We are also currently considering leveraging Bayesian optimization [9], [10] and reinforcement learning for addressing the third challenge, but we want to confirm that our approach here is correct. To address the challenges, we first evaluated the query performance of TPC-DS on multiple mixed query engines (Spark and Tez) and JVMs (OpenJDK and J9). On the basis of benchmark results, we found that unsuitable systems lead to a drawback of over 10 times. Then, we analyzed where the drawback comes from with system profiling. Finally, we propose a classification model for making the best combination of systems on the basis of the analysis. The model utilizes generated query plan (DAG) information generated by Spark and Tez. We examined several machine learning algorithms and then confirmed that the proposed classifier can reduce the execution time by 50% in total. In summary, we make the following contributions in this paper.

- We characterize the query performance of TPC-DS with a combination of SQL-on-Hadoop systems and JVMs.
- We analyze the fundamental causes of performance gain and drawback found through our experiments that would be helpful for not only making a classification model but also improving the core of JVMs and SQL-on-Hadoop systems.
- We provide a classification model based on DAG characteristics generated by both query engines and determine the classifier performance with all TPC-DS queries.

II. Background

A. Spark SQL and Spark

Apache Spark is an open source in-memory-oriented data processing framework for big data processing, and it has been widely used as a general-purpose distributed engine [11]. By keeping as much data in-memory as possible without writing intermediate data, Spark performs much better than Hadoop.

Spark SQL [4] is a runtime for query processing on top of Spark, and it introduces a query optimizer called "Catalyst." Catalyst makes an optimized physical task execution plan (DAG) and generates optimized Java bytecode to speed up execution. By exploiting hardware performance with SIMD instructions and building a more efficient memory management model and serialization format, Spark SQL lets applications run at the speed offered by the bare metal. Spark SQL has the compatibility to run Hive Query, i.e., HiveQL, which is shown in a later section, so the same HiveQL runs on Spark SQL without modification. Spark SQL is highly optimized for utilizing the Parquet file, which is a columnar format. Columnar data representation has many advantages over the row-based format, for example, column pruning and vectorized processing.

B. Hive/LLAP and Tez

Apache Hive is one of the most widely used query engines built on top of Hadoop and provides a SQL-like query language called "HiveQL." Hive was designed for batch-style large data processing with a MapReduce job, which is compiled from HiveQL. Apache Tez [12] is a data-flow-oriented distributed-processing framework, and it was developed as a successor to the MapReduce engine. The data processing flow is described as a DAG similar to Spark.

Recently, there have been attempts to make Hive effective in supporting executing-latency aware interactive ad-hoc queries with many features such as cost-based optimization (CBO), columnar storage, vectorization, and asynchronous I/O threads. The Optimized Row Columnar (ORC) is a highly efficient file format for Hive. Hive introduces Calcite into its CBO engine. Calcite is an open-source CBO, and it has more than 50 query optimization rules. Moreover, Hive introduces low latency analytical processing (LLAP), which is a key feature for using these optimizations in the latest Hive. LLAP resides in each node as a long-lived daemon, and it provides an in-memory columnar table cache and well-optimized operator pipeline exploiting SIMD instructions.

C. Java Virtual Machines

As Hadoop is the de facto middleware in big-data processing, JVM takes an important role as a runtime of these eco-systems because Hadoop itself and related frameworks are written in Java or Scala. Due to the recent in-memory processing shift on these frameworks, JVM tuning is even more necessary to exhibit optimum performance. Similar to the other Java workloads, it is well known that GC greatly affects application performance on big-data middleware including Spark [13]. Besides GC being important in terms of memory management, optimizing execution code by using a JIT compiler is becoming even more important to enable higher performance for in-memory query processing on recent SQL-on-Hadoop systems.

First, JIT generally optimizes bytecode on the basis of two different goals; one is to minimize startup time, and the other is to maximize the performance of JIT compiled code with method profiling. Since JIT compilation itself and hot method profiling require CPU resources, it is important to choose an adequate compilation policy that takes into consideration the balance between compilation cost and execution time. Recent JVMs offer tiered compilation, which makes it possible to balance compilations at runtime. OpenJDK introduces two

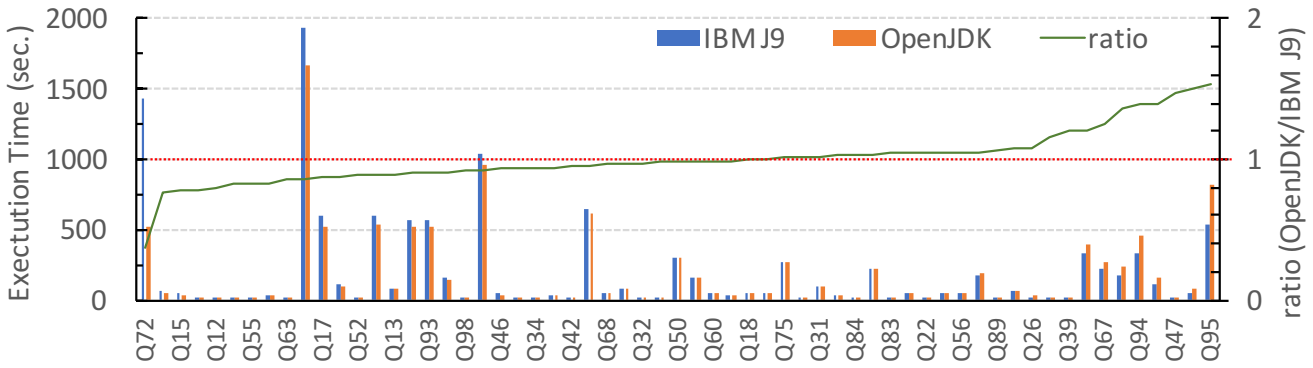


Fig. 1. Performance comparison on TPC-DS (SF = 500) of OpenJDK and J9 on Spark

tiered compilation, called "C1" and "C2," and J9 does six tiered compilation [14].

Second, vectorized code generation may further improve query processing by exploiting hardware performance with SIMD instructions. As some studies [15], [16] have shown the benefits of vectorization in query processing, recent SQL-on-Hadoop systems are also aimed at utilizing the auto-vectorization feature of a JIT compiler. The columnar data format also encourages systems to move into vectorization because it helps the JIT to perform vectorization more effectively by loading the same type of values in a memory array.

Third, a JIT compiler offers intrinsics for several Java methods such as *System.arraycopy()*. Without having a compilation, JIT compiler can replace these methods with architecture-optimized code. Both OpenJDK and J9 offers intrinsics for performance improvement.

III. Performance Evaluation

In this section, we will identify potential gains in modifying all of four combinations of execution frameworks (Spark and Tez) and JVMs (OpenJDK and J9) in TPC-DS queries.

We conducted all experiments on a single node of Power System S824L, which is equipped with two 3.3-GHz POWER8 processors, 1 TB of RAM, and 5 TB of flash storage. Each processor has 12 cores, while each core has 64 KB of L1 cache, 512 KB of L2 cache, and shares 96 MB of L3 cache. The system manages a total of 192 hardware threads with 8 hardware threads per core and runs on Ubuntu 16.04 (kernel: 4.4.0-31-generic). We used Hadoop 2.7.2, Spark 2.1.0, Tez 0.9.0, and Hive 2.2.0 and prepared two JVMs, IBM J9 JVM (1.8.0 SR4 FP2) and OpenJDK8, which are based on the same Java class libraries (jdk8u121-b13).

Regarding the dataset, we generated 500-GB TPC-DS (scale factor = 500) raw table data using hive-testbench[1], which were loaded in Parquet and ORC formats with gzip compression on HDFS. Spark SQL and Hive/LLAP can process Parquet and ORC format tables, but Spark SQL is optimized for Parquet and Hive/LLAP for ORC, respectively. To not be biased toward one system, we conducted a query-performance evaluation with Parquet for Spark SQL and with ORC for Hive/LLAP. We tested all 68 TPC-DS queries that are available in hive-testbench.

Due to the framework difference between Spark and Tez, we could not apply the same configuration for both frameworks, but we used the same amount of resources as possible for fair comparison. We prepared 12 worker threads and a 192-GB heap for Spark SQL/Spark and 12 worker threads, 12 I/O threads, a 96-GB heap, and 96-GB LLAP cache for Hive/LLAP/Tez. To simplify the terminologies for the runtimes, we denote Spark SQL/Spark as Spark and Hive/LLAP/Tez as Tez.

A. J9 or OpenJDK - Spark Case

First, we will show the potential gain for different JVMs on Spark. Figure 1 shows the query-response time (shown with bars) and the performance gap (shown with a line) between OpenJDK and J9 on Spark. The ratio means that OpenJDK was faster than J9 when it was smaller than 1. This result includes 62 out of 68 queries, while the remaining 6 queries failed due to a query-format error. Focusing on both ends, Q72 finished 2.7 times faster with OpenJDK, and Q95 finished 1.5 times faster with J9. In summary, OpenJDK ran faster for 35 queries, and J9 ran faster for the remaining 27 queries.

B. J9 or OpenJDK - Tez Case

Next, we conducted the same experiments on Tez while changing JVMs. A total of 65 queries successfully ran, except the remaining 3 queries. Similar to the Spark results in Figure 1, the query performance varied with the query, but the slow queries on Spark were not always slow on Tez, which we discuss later. Since the graph characteristics were almost the same, we omitted the graph of the performance results for Tez due to page limitations. OpenJDK was 1.56 times faster for Q87 than J9, and J9 also was up to 1.74 times faster for Q84 than OpenJDK.

C. Spark or Tez

We then evaluated the query performance for different frameworks (Spark and Tez) with OpenJDK or J9. Figure 2

[1]https://github.com/hortonworks/hive-testbench

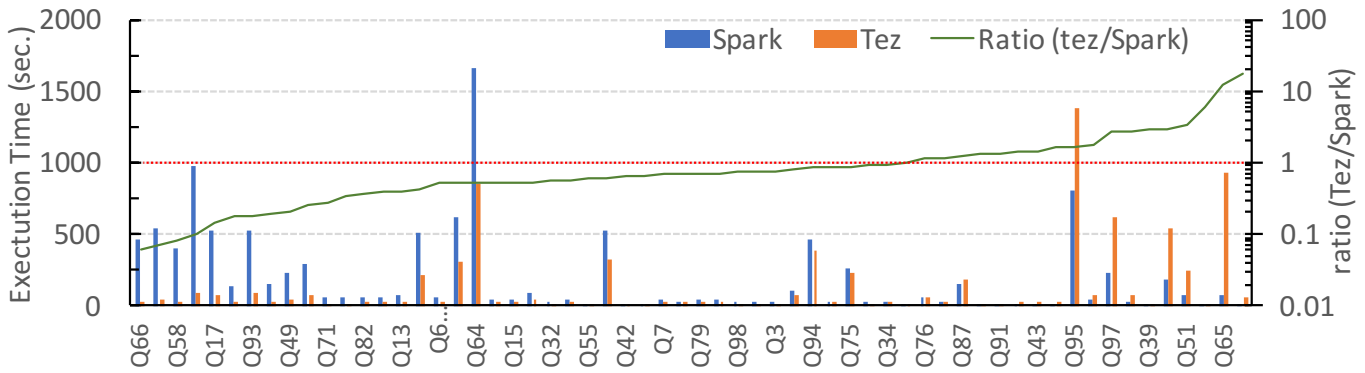


Fig. 2. Performance comparison on TPC-DS (SF = 500) of Spark and Tez on OpenJDK

TABLE I
OPTIMAL COMBINATIONS OF QUERY ENGINE AND JVM

	J9	OpenJDK	total
Spark	13	6	19
Tez	22	19	41
total	35	25	60

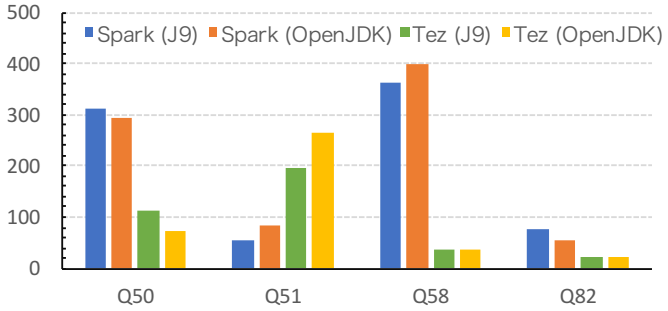
shows the results of all 60 successfully finished queries with OpenJDK. Since using J9 had a similar tendency, a graph of the performance results is not shown to save space. Tez performed better for around 40 queries than Spark, and Spark performed better for around 20 queries. Compared with a change in JVMs, as shown in previous sections, selecting the wrong framework leads to a larger drawback; a suitable framework is up to 10 times faster than Spark or Tez.

Finally, we compared the query response time for all combinations (Spark, Tez, OpenJDK, and J9) and summarized what combination is best for all 60 queries. Table I shows how many queries were categorized into each combination. Focusing on the Q13 response time, for example, Q13 finished in 73.5 sec, 81.9 sec, 29.7 sec, and 25.1 sec on Spark with OpenJDK, Spark with J9, Tez with OpenJDK, and Tez with J9, respectively. Thus, we concluded that Tez with J9 is the best combination for Q13.

IV. PERFORMANCE ANALYSIS

As shown in the previous section, we can improve query performance by selecting the best combination of execution framework and JVM for a query. However, we still do not understand where the performance gain comes from, what characteristics make the difference, and when to choose a different framework or JVM. In terms of making a model for selecting the best combination, it is important to identify reasonable features from many characteristics including frameworks, JVMs, and queries. In this section, we will analyze the details of the performance difference in frameworks and JVMs on the basis of various metrics, such as Java and JVM method profiling and generated query plan (DAG) analysis, and then determine which features impact performance.

Fig. 3. Comparison of target queries on mixed frameworks and JVMs



There is not enough space to show the analysis results for all TPC-DS queries, so we focused on several representative queries that have interesting characteristics. The selected queries were Q50, Q51, Q58, and Q82. Figure 3 shows the performance for these queries with different combinations of framework and JVM. For example, Spark(J9) denotes the query response time running on Spark as a framework and using J9 as a JVM. Q51 is a representative example to show that J9 has a 30-40% advantage over OpenJDK for both query engines and that Spark is 3.5 times faster than Tez. Q50 and Q82 indicate that OpenJDK is 30-35% faster than J9 on Tez and Spark, respectively. Q50, Q58, and Q82 indicate that Tez is 2-10 times faster than Spark.

A. Analyzing Differences in JVMs on Spark

First, we analyzed the performance difference between J9 and OpenJDK on Spark. Table II describes the characteristics of executing the DAGs of Q51 and Q82 generated by Spark. These characteristics are also candidate features for a classification model that we will discuss later. Reduce stages have shuffling for join operations, and the stages also write intermediate data for shuffling. Comparing the DAGs of the two queries, we conclude that Q51 is characterized as being input-data heavy, shuffle-data-light, and having many shuffle

TABLE II
DAG ANALYSIS OF Q51 AND Q82 ON SPARK

	Q51	Q82
# of map stages	2	2
# of reduce stages	6	2
# of bcast hash joins	2	2
# of sort-merge joins	1	1
# of grpby/sort	5	1
input read size (parquet,zlib)	6.0GB	2.5GB
shuffle output size (lz4)	1.0GB	5.6GB

stages, and Q82 is characterized as being input-data-light, shuffle-data heavy, and having a few shuffle-stages.

We then analyzed hot methods for Java, JVM, and a native library while running Q51 and Q82, profiled by Oprofile. Table III represents the overall profile of each category (top half) and a breakdown of the profiles in the Java category (bottom half). JVM includes JVM managed methods like GC and JIT, nativelib includes mainly compression libraries like zlib and lz4, and Java includes application and framework methods. In the top half, one notable difference exists in JVM; OpenJDK spent much more CPU for JVM than J9 because GC threads in OpenJDK waited for GC tasks in a spin loop. However, spin wait does not affect performance as long as CPU resources remain.

Moving to the bottom half of the breakdown of Java, each of the categories (catalyst, java and spark lib, crc32c, lz4-java, and unsafe) corresponds to query processing, framework and I/O processing, crc32c validation for input data, decompression, and memory management with *sun.misc.Unsafe*, respectively. One notable difference here is the unsafe ratio. *sun.misc.Unsafe.copyMemory()* is frequently used in Spark shuffle; however, there exists a constant overhead due to calling this method compared with J9 because OpenJDK does not provide an optimized intrinsic for *Unsafe.CopyMemory*. Furthermore, the overhead of JNI calls was not negligible when many tasks and stages existed. Q51 had many reduce tasks consisting of 200 tasks for shuffling, but the total amount of shuffling was relatively smaller than that of Q82, so a JVM-management overhead accumulated between stages. In comparison, J9 spent around 10% more CPU cycles for java and spark lib while running Q82. As can be seen in the detailed profile, J9 spent much more time while writing data to an intermediate file for a later shuffle.

In summary, for SQL workloads on Spark with J9 and OpenJDK, J9 has an advantage in complex multi-tier queries that have many stages including shuffle, and OpenJDK has an advantage in simple stage-less queries that write a shuffle-output file larger than the input-file size. This tendency was common for Q50 and Q58.

B. Analyzing Differences in JVMs on Tez

Next, we compared the difference between J9 and OpenJDK on Tez for Q50 and Q51. Table IV summarizes the overall profile result. Compared with the MapTask in Tez, J9 always performed better than OpenJDK. MapTask in Tez consists of several phases like data loading, sorting, and writing in a pipeline fashion. In the MapTask for Q51, for example, J9 took 7 sec on average, whereas OpenJDK took 21 sec on average. Focusing on the details of Java methods, a hotspot does not exist in applications but in data copy and serialization. The ratio for OpenJDK was higher than that for J9.

Moving on to the best case of OpenJDK for Q50, OpenJDK was also behind for MapTask as well as Q51, but this drawback did not result in a difference in performance because the input data were not that large. Instead, ReduceTasks spent a lot of time for Q51, which took 106 sec with J9 and about 60 sec with OpenJDK. As shown in the method-profile results in Table IV, J9 spent three times the CPU cycles for kallsyms compared with OpenJDK. Comparing CPU cycles, J9 spent three times more system CPU time and used only half of the CPU time compared with OpenJDK. For Tez, many threads were managed in each vertex of the computational DAG for handling various tasks. Most of these threads were related to shuffle tasks such as merge, sort, file writing, and file fetching. Runtime automatically prepares threads corresponding to the logical CPU cores of a system. By checking the system utilization, J9 caused much more context switches than OpenJDK.

In summary, for SQL workloads on Tez with J9 and OpenJDK, J9 has an advantage in executing MapTask-heavy workloads, and OpenJDK has an advantage in ReduceTask-heavy workloads.

C. Spark or Tez - Analyzing Differences in Framework

Finally, we analyzed the difference in performance between Spark and Tez. As shown in Figure 3, Spark was faster than Tez for Q51, and Tez was faster for the other three queries.

TABLE III
BREAKDOWN OF OVERALL AND JAVA CPU CYCLES ON SPARK

		Q51		Q82	
	category	J9	OpenJDK	J9	OpenJDK
Overall	others	18.3%	15.4%	16.3%	9.77%
	nativelib	15.6%	11.8%	4.43%	5.53%
	JVM	15.7%	28.3%	5.49%	10.5%
	Java	50.4%	44.5%	73.8%	74.2%
Java detail	catalyst	25.6%	13.8%	15.7%	13.0%
	java io	11.8%	14.6%	12.5%	20.0%
	spark lib	6.71%	7.67%	35.4%	19.5%
	crc32c	2.49%	1.95%	0.58%	0.9%
	lz4-java	3.75%	4.66%	9.59%	18.5%
	Unsafe	0.0001%	1.78%	0.0003%	2.2%

TABLE IV
BREAKDOWN OF OVERALL AND JAVA CPU CYCLES ON TEZ

		Q50		Q51	
	category	J9	OpenJDK	J9	OpenJDK
Overall	others	18.4%	8.82%	4.30%	3.90%
	nativelib	2.83%	4.18%	2.20%	2.04%
	JVM	10.0%	25.9%	12.0%	7.06%
	Java	68.7%	61.1%	81.5%	87.0%
Java detail	hive ql	28.0%	22.2%	15.2%	8.97%
	tez lib	3.37%	2.81%	29.2%	12.0%
	orc	21.7%	17.1%	0.92%	0.89%
	hadoop	8.25%	5.49%	26.2%	44.5%
	java	7.24%	13.3%	9.46%	20.6%

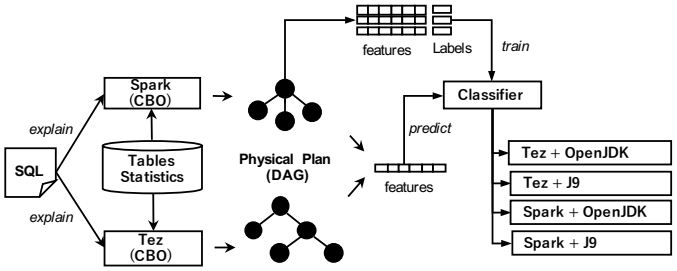


Fig. 4. Overview of proposed classifier (training and prediction)

We will look at the Q51 case in more detail. In terms of the form of the query plan, the formation was almost the same, but one difference came from the implementation of map-side join. Map-side join is a technique for reducing shuffling data in order to combine a small table with a big table. In Spark, the driver program loads small tables and then pushes them to the executor JVMs. In Tez, instead, executor JVMs directly pull the small tables. At this point, the DAG is slightly different, but this does not impact performance.

Comparing the query plan inside, the filtering rule for a big table was different. Spark was more aggressively optimized for pruning unused rows, and the total number of loaded lines was 366; however, that for Tez was 8116. As a result, the intermediate data size for shuffling for the successor ReduceTask was quite different; 687 MB for Spark and 3.6 GB for Tez. In contrast, Tez performed better than Spark for Q50, Q58, and Q82 with another filter optimization, i.e., Bloom filter. Since a Bloom-filter-based filter rule is implemented in Tez only, it is effective in data pruning if the feature is enabled. Back to Q51 again, Spark performed better than Tez because the rule does not work in Tez. In summary, the filtering rule in the first data loading map phase makes a difference in performance.

V. Classifier for Engine and Runtime Selection

A. Training Methodology

As discussed in Section IV, we characterized what features make a difference in performance by analyzing observation results and comparing the details of implementation for each query engine and JVM. To utilize these features for classifier training and prediction, we propose using a generated DAG from Spark and Tez as a training data set because a DAG already contains the candidate features, e.g., the number of stages, maps, and reduces, input and shuffle data size, and filtering rules, we listed in Section IV.

Figure 4 represents an overview of the training and prediction steps of our classifier. Since each CBO generates an optimized physical plan (DAG) on the basis of table statistics and their own strategies, we can extract the key features from them and then make a training feature vector per each query. As shown in Table I, we already had training labels for all queries, so we can perform supervised learning for this classifier problem. For the prediction phase, we execute the *explain* command for a query. By using *explain*, we can get an optimized DAG without having to actually execute a query. By passing the feature vector to a classifier, we can predict the best suitable combination of engine and runtime for a query.

TABLE V
Accuracy scores of the classifiers

	binary classifier		multinomial classifier	
	mean	stddev (+/-)	mean	stddev (+/-)
kNN	0.65	0.08	0.43	0.04
Decision Tree	0.69	0.17	0.34	0.16
SVM	0.72	0.10	0.39	0.08
Random Forest	0.72	0.02	0.46	0.06

B. Evaluating Classification Model

We generated our classification model on the basis of the TPC-DS results shown in Section III. We extracted a total of 69 features from 2 DAGs, that is, 34 features from Spark DAG and 35 features from Tez DAG. We prepared two classifier models in this experiment. The binary classifier focused on selecting the query engines: Spark or Tez. The multinomial classifier focused on selecting the best combination of engine and JVM from four patterns.

Table V shows the accuracy scores of the classifiers with four different machine learning algorithms: k-nearest neighbor (kNN), decision tree, support vector machine (SVM), and Random Forest. To prevent overfitting, we conducted k-fold cross validation for all four algorithms by dividing the dataset into 80:20 for training and testing. In our experiments, random forest achieved a higher accuracy score than the other three algorithms; the score was 0.72 for the binary classifier and 0.46 for the multinomial classifier. Compared with the simple decision tree, random forest performed better with ensemble methods. We can improve accuracy scores more by searching for well-suited hyperparameters. In the case of Random Forest, the accuracy score was improved up to 0.82 for the binary classifier and up to 0.54 for the multinomial classifier by changing the number of tree depth and ensemble trees. The accuracy score of the multinomial classifier seemed to be not that high compared with the score of the binary classifier because it included a slight mis-prediction in JVM selection. However, this mis-prediction would not cause a big performance difference in practice compared with the query engine. As a consequence, we adopt Random Forest for our classifier.

To inspect which features actually contribute to our Random Forest based classifier, we then evaluated the feature importances of the classifier, which is shown in Figure 5. We only listed up the top 15 features. The prefix "S" means the feature comes from Spark, and prefix "T" means the feature comes from Tez. As we expected, the basic DAG features, e.g., the number of stages, were more important than others. Map-side join related features, i.e., S_BcastExch and T_MapjoinM, and the features of the used tables, e.g., store_sales and item, are also affected the classifier. In contrast, some features, e.g., using bloom filter or outer join, were not so important because they are already covered by the basic DAG features.

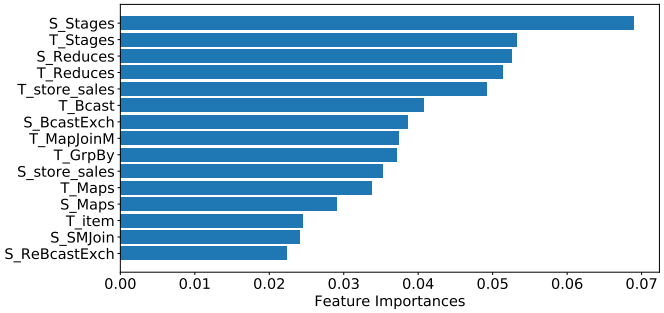


Fig. 5. Feature importances of our Random Forest based classifier

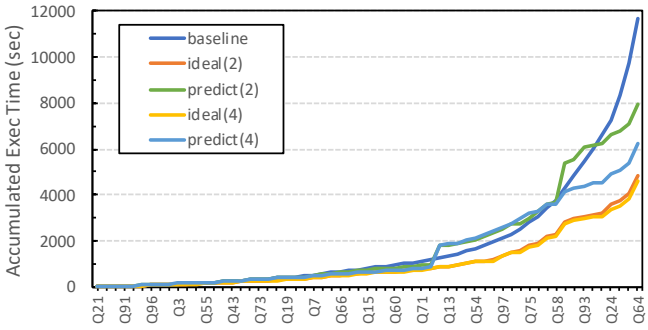


Fig. 6. Performance of Classification Models (baseline=Spark/J9, predict(2)=binary classifier, predict(4)=multinomial classifier)

Finally, we evaluated how much performance improvement we can achieve if these classification models are used. Figure 6 compares the total accumulated execution time through all TPC-DS queries. The baseline represents the query response time with a fixed combination, that is, Spark/J9. The data are sorted in ascending order. The "ideal" lines represent the performance with the best selection, and the subscript, e.g., 2 or 4, means the number of classification classes. The "predict" lines also represent the performance with our classification models. Comparing the baseline with the ideal, the baseline took about 12,000 sec until all query executions were finished, while the ideal took 4,600 sec. There was not that much of a difference between ideal(2) and ideal(4). In terms of prediction, predict(2) made big mistaken decisions for two queries. However, it made good predictions for several heavy queries such as Q93, Q24, and Q64, so it reduced execution time by 35% in total compared with the baseline. Predict(4) achieved more of a performance improvement than did predict(2), and it reduced the execution time by 50% in total.

VI. Related Work

Performance Characterization: Analyzing performance characteristics with typical benchmarks is always important because such knowledge helps in determining what system is best and where the fundamental system bottleneck is. Thus, several studies have attempted to characterize SQL-on-Hadoop systems [7], [17], [8], and few studies have characterized systems from a JVM perspective [18], [19]. Floratou et al. [7] compared three systems (Hive on MapReduce, Hive on Tez, and Impala) and summarized their performances by conducting the TPC-H benchmark. Shi et al. [17] compared MapReduce and Spark by using various typical workloads such as word count, page rank, sort, and k-means. Qin et al. [8] evaluated five recent systems (Hive, Spark, Presto, Imparam and Drill). From the JVM point of view, in papers [18], [19], Spark-performance optimization was characterized and discussed, especially with heap sizing and GC improvement. Almost all studies focused on Hive and SparkSQL, but there has been no study addressing the recent Hive LLAP feature. Moreover, there has been no comparison of multiple JVM implementations, though their differences result in large performance gaps.

Performance Model: Cost-based performance models have been widely investigated for the Hadoop eco-system [20], [21], and these models are helpful in accurately predicting performance. One drawback of this approach is that precise knowledge on a system is required to construct a concrete model. Thus, machine-learning-based models have also been investigated. Luo et al. [22] used a support vector machine (SVM) and artificial neural network (ANN) to construct a performance model for Spark. Gibilisco et al. [23] proposed a mixed approach for a Spark-performance model, which is based on training with multiple polynomial regression models and considering per-stage characteristics of a Spark DAG. Kewen et al. [24] constructed an analytical model to estimate the effort of interference among multiple Spark jobs, and Nhan et al. [25] demonstrated an analytical model for understanding the effect of Spark configurations on several Spark applications.

Combination of Multiple Engines: Several studies have proposed mixing engine/library approaches recently to exploit the advantage of different engines for big-data-analytics platforms. Musketeer [26] decouples frontend frameworks from backend engines, translates a workflow into an intermediate representation (IR), and then generates code for the best suitable backend by applying several optimizations to the Musketeer IR DAG. By expressing an application as an IR and facilitating cross-library optimization, Weld [27] performs competitively with hand-tuned state-of-the-art code on various systems such as Spark and TensorFlow. Hybrid query-engine approaches, such as MISO [28], Octopus [29], and MuSQLE [30], have been extensively investigated and determined to perform best for queries. On the basis of analytical or heuristic approaches, they adaptively select the best or multiple query engines from Spark, Hadoop, PostgreSQL, MySQL, and so on. These studies had a similar motivation to ours in terms of exploiting the power of backend engines. In addition to considering multiple query engines, we addressed the combination of query engine and JVM runtime.

VII. Conclusion

We evaluated the performance of TPC-DS on several combinations of SQL-on-Hadoop systems (Spark and Tez) and JVMs (OpenJDK and J9) and analyzed the performance characteristics of each system through method profiling and DAG analysis. We demonstrated that system mismatch leads to a huge drawback depending on the query characteristics; using a different query engine would be up to 10 times worse, and using a different JVM would be up to 3 times worse. Then, we discussed classification models based on a DAG generated by each query engine. By executing several machine learning algorithms, it was found that random forest can make a prediction more correctly than others. As a consequence, the proposed random forest based classifier reduced the execution time by 50% in total while running all TPC-DS queries by selecting better combinations of query engine and JVM.

References

[1] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, May 2010, pp. 1–10.

[2] M. Zaharia, "How spark usage is evolving in 2015," *Spark Summit Europe 2015*, 2015.

[3] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive - a petabyte scale data warehouse using hadoop," in *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, March 2010, pp. 996–1005.

[4] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1383–1394.

[5] M. Kornacker, A. Behm, and V. e. a. Bittorf, "Impala: A modern, open-source sql engine for hadoop," in *Proceedings of 7th Biennial Conference on Innovative Data Systems Research (CIDR '15)*, Asilomar, California, USA, 2015.

[6] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber, "Presto: Distributed machine learning and graph processing with sparse matrices," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. ACM, 2013, pp. 197–210.

[7] A. Floratou, U. F. Minhas, and F. Özcan, "Sql-on-hadoop: Full circle back to shared-nothing database architectures," *Proc. VLDB Endow.*, vol. 7, no. 12, pp. 1295–1306, Aug. 2014.

[8] X. Qin, Y. Chen, J. Chen, S. Li, J. Liu, and H. Zhang, "The performance of sql-on-hadoop systems - an experimental study," in *2017 IEEE International Congress on Big Data (BigData Congress)*, June 2017, pp. 464–471.

[9] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 469–482.

[10] V. Dalibard, M. Schaarschmidt, and E. Yoneki, "BOAT: Building auto-tuners with structured bayesian optimization," in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW '17. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2017, pp. 479–488.

[11] M. Armbrust, T. Das, A. Davidson, A. Ghodsi, A. Or, J. Rosen, I. Stoica, P. Wendell, R. Xin, and M. Zaharia, "Scaling spark in the real world: performance and usability," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1840–1843, 2015.

[12] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache tez: A unifying framework for modeling and building data processing applications," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1357–1369. [Online]. Available: http://doi.acm.org/10.1145/2723372.2742790

[13] E. Kaczmarek and L. Yi, "Taming gc pauses for humongous java heaps in spark graph computing," *Spark Summit 2015*, 2015.

[14] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan, "Java just-in-time compiler and virtual machine improvements for server and middleware applications," in *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium*, ser. VM'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 12–12.

[15] M. Zukowski, M. van de Wiel, and P. Boncz, "Vectorwise: A vectorized analytical dbms," in *2012 IEEE 28th International Conference on Data Engineering*, April 2012, pp. 1349–1350.

[16] A. Costea, A. Ionescu, B. Raducanu, M. Switakowski, C. Barca, J. Sompolski, A. Luszczak, M. Szafranski, G. de Nijs, and P. Boncz, "VectorH: Taking sql-on-hadoop to the next level," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: ACM, 2016, pp. 1105–1117.

[17] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan, "Clash of the titans: Mapreduce vs. spark for large scale data analytics," *Proceedings of the VLDB Endowment*, vol. 8, no. 13, pp. 2110–2121, 2015.

[18] E. Kaczmarek and L. Yi, "Taming gc pauses for humongous java heaps in spark graph computing," in *Spark Summit 2015*, 2015, https://spark-summit.org/2015/events/taming-gc-pauses-for-humongous-java-heaps-in-spark-graph-computing/.

[19] T. Chiba and T. Onodera, "Workload characterization and optimization of tpc-h queries on apache spark," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 112–121.

[20] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A self-tuning system for big data analytics," in *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, 2011, pp. 261–272.

[21] X. Lin, Z. Meng, C. Xu, and M. Wang, "A practical performance model for hadoop mapreduce," in *2012 IEEE International Conference on Cluster Computing Workshops*, Sept 2012, pp. 231–239.

[22] N. Luo, Z. Yu, Z. Bei, C. Xu, C. Jiang, and L. Lin, "Performance modeling for spark using svm," in *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, Nov 2016, pp. 127–131.

[23] G. P. Gibilisco, M. Li, L. Zhang, and D. Ardagna, "Stage aware performance modeling of dag based in memory analytic platforms," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, June 2016, pp. 188–195.

[24] K. Wang, M. M. H. Khan, N. Nguyen, and S. Gokhale, "Modeling interference for apache spark jobs," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, June 2016, pp. 423–431.

[25] N. Nguyen, M. M. H. Khan, Y. Albayram, and K. Wang, "Understanding the influence of configuration settings: An execution model-driven framework for apache spark platform," in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, June 2017, pp. 802–807.

[26] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand, "Musketeer: All for one, one for all in data processing systems," in *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. New York, NY, USA: ACM, 2015, pp. 2:1–2:16.

[27] S. Shoumik Palkar, J. J. Thoms, A. S. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, and M. Zaharia, "Weld: A common runtime for high performance data analytics," in *Proceedings of 8th Biennial Conference on Innovative Data Systems Research (CIDR '17)*, ser. CIDR '17, Chaminade, CA, USA, 2017.

[28] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey, "MISO: Souping up big data query processing with a multistore system," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 1591–1602.

[29] Y. Chen, C. Xu, W. Rao, H. Min, and G. Su, "Octopus: Hybrid big data integration engine," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, Nov 2015, pp. 462–466.

[30] V. Giannakouris, N. Papailiou, D. Tsoumakos, and N. Koziris, "Musqle: Distributed sql query execution over multiple engine environments," in *2016 IEEE International Conference on Big Data (Big Data)*, Dec 2016, pp. 452–461.