LOCAT: Low-Overhead Online Configuration Auto-Tuning of Spark SQL Applications

Jinhan Xin
jh.xin@siat.ac.cn
Shenzhen Institute of Advanced Technology (SIAT), Chinese Academy of Sciences (CAS)
Shenzhen, Guangdong, China
University of Chinese Academy of Sciences (UCAS)
Beijing, China

Kai Hwang
hwangkai@cuhk.edu.cn
The Chinese University of Hong Kong, Shenzhen
Shenzhen, Guangdong, China

Zhibin Yu*
zb.yu@siat.ac.cn
Shenzhen Institute of Advanced Technology (SIAT), Chinese Academy of Sciences (CAS)
Shenzhen, Guangdong, China
Shuhai Lab, Shenzhen Huawei Cloud Computing Co.,Ltd.
Shenzhen, Guangdong, China

ABSTRACT

Spark SQL has been widely deployed in industry but it is challenging to tune its performance. Recent studies try to employ machine learning (ML) to solve this problem, but suffer from two drawbacks. First, it takes a long time (high overhead) to collect training samples. Second, the optimal configuration for one input data size of the same application might not be optimal for others.

To address these issues, we propose a novel Bayesian Optimization (BO) based approach named LOCAT to automatically tune the configurations of Spark SQL applications online. LOCAT innovates three techniques. The first technique, named QCSA, eliminates the configuration-insensitive queries by Query Configuration Sensitivity Analysis (QCSA) when collecting training samples. The second technique, dubbed DAGP, is a Datasize-Aware Gaussian Process (DAGP) which models the performance of an application as a distribution of functions of configuration parameters as well as input data size. The third technique, called IICP, Identifies Important Configuration Parameters (IICP) with respect to performance and only tunes the important ones. As such, LOCAT can tune the configurations of a Spark SQL application with low overhead and adapt to different input data sizes.

We employ Spark SQL applications from benchmark suites $TPC-DS$, $TPC-H$, and $HiBench$ running on two significantly different clusters, a four-node ARM cluster and an eight-node x86 cluster, to evaluate LOCAT. The experimental results on the ARM cluster show that LOCAT accelerates the optimization procedures of the state-of-the-art approaches by at least 4.1× and up to 9.7×; moreover, LOCAT improves the application performance by at least 1.9× and up to 2.4×. On the x86 cluster, LOCAT shows similar results to those on the ARM cluster.

*Zhibin Yu is the corresponding author. An extended version of this paper is at arXiv:2203.14889[cs.DC]

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA.

ACM ISBN 978-1-4503-9249-5/22/06.
https://doi.org/10.1145/3514221.3526157

CCS CONCEPTS

• **Information system applications** → *Computing platforms*; • **Information Systems** → Data management systems; • **Computing methodologies** → Distributed computing methodologies.

KEYWORDS

big data, in-memory computing, Spark, Spark SQL

ACM Reference Format:
Jinhan Xin, Kai Hwang, and Zhibin Yu. 2022. LOCAT: Low-Overhead Online Configuration Auto-Tuning of Spark SQL Applications. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3514221.3526157

1 INTRODUCTION

Big data query systems such as Hive [49], Presto [42], and Spark SQL [4] have been widely deployed in industry to mine valued information from massive data efficiently [41]. As a higher level library on top of Apache Spark [60], Spark SQL not only inherits Spark's excellent big data processing capabilities, but also provides support for query-like large-scale data analysis, such as OnLine Analytical Processing (OLAP) [16, 35].

However, it is challenging to tune the configuration parameters of a Spark SQL application for optimal performance because of two reasons. First, the lower layer Spark and the upper layer Spark SQL both have a number (e.g.,> 20) of configuration parameters. Not only the ones for Spark SQL (e.g., *spark.sql.shuffle.partitions*) itself, but also those for Spark (e.g., *spark.executor.memory*) significantly affect the performance of a Spark SQL application. For example, the parameter *spark.executor.memory* specifies the amount of memory used by an executor process [46]. Too large value of it may cause a long garbage collection time [14] pausing the application whereas too small value may even lead to out of memory (OOM) errors [32]. Therefore, tuning the larger number of parameters for optimal performance of a Spark SQL application is difficult. Second, the configuration parameters within the same layer as well as from different layers may intertwine with each other in a complex way with respect to performance, further troubling the performance tuning of a Spark SQL application.

Recent studies propose to leverage machine learning (ML) to tune the configurations for Spark programs [32, 58] and database

systems [33, 63]. However, these studies have two drawbacks. **First**, it takes a long time to collect training samples, which is inconvenient in practice. The long time stems from four factors. (1) The number of training samples is large (e.g., 1000 — 10000), which is the nature of ML-based approaches. (2) The time used to collect each training sample of an application is typically long (e.g., several minutes) because it is collected by running the application on a real cluster with a randomly generated configuration. (3) A Spark SQL application typically consists of a number (e.g., 20) of queries. The more queries in a Spark SQL application generally make it take longer time to execute and in turn longer time to collect one training sample. (4) ML-based approaches generally need more training samples when tuning more configuration parameters.

Second, most ML-based approaches can not adapt to the changes of input data sizes of a Spark SQL application. That is, a configuration making a Spark SQL application achieve the optimal performance for one input data size might not produce optimal performance for another input data size. This makes the same application need to be re-tuned when its input data size is changed, which is time-consuming. However, customers typically do not change their Spark SQL applications frequently while definitely often change the input data size of the same application.

To address these issues, we propose a novel approach dubbed LOCAT to automatically tune the configurations of a Spark SQL application online. LOCAT's first key innovation is that we observe an important as well as interesting finding: *different queries in a Spark SQL application respond to configuration parameter tuning with significantly different sensitivity*. Some queries of an application are insensitive to the parameter tuning at all, and we therefore call them *configuration-insensitive* queries. Based on this finding, we remove the configuration-insensitive queries from a Spark SQL application when we run the application with random configurations to collect training samples. As such, the sample collection time can be dramatically reduced.

The second key innovation is that we propose a Datasize-Aware Gaussian Process (DAGP) to take the input data size in addition to the configuration parameters of a Spark SQL application into consideration as tuning the configuration parameters. In contrast, other Gaussian Process (GP) based approaches such as CherryPick [2] only consider the configuration parameters, which needs to perform the time-consuming parameter re-tuning when an application's input data size is changed. The third innovation is that we propose to identify the important configuration parameters of a Spark SQL application and in turn only tune them in BO (Bayesian Optimization) iterations. We name this technique IICP. Generally, tuning more parameters takes more iterations to find the optimal configuration for an application by BO. IICP therefore takes less iterations and in turn shorter time to find the optimal configuration.

In particular, this paper makes the following contributions.

- We find that some queries of a Spark SQL application are insensitive to configuration parameter tuning with respect to performance by Query Configuration Sensitivity Analysis (QCSA). We therefore remove these queries from the application when we run it to collect training samples.
- We propose to leverage Gaussian Process (GP) to model the relationship between performance and the input data size of a Spark SQL application in addition to the configuration parameters. As such, our approach can adapt to different input data sizes of the same application.
- We propose to identify the important configuration parameters (IICP) of a Spark SQL application and only tune these parameters in order to reduce the tuning time.
- By putting it all together, we develop an online configuration parameter tuning approach for Spark SQL applications with low overhead, named LOCAT.
- We employ Spark SQL applications from $TPC-DS$, $TPC-H$, and $HiBench$ running on two different clusters — a four-node ARM cluster and an eight-node x86 cluster — to evaluate LOCAT. The experimental results on the ARM cluster show that LOCAT accelerates the optimization procedures of the state-of-the-art approaches by at least 4.1× and up to 9.7×; moreover, LOCAT improves the application performance by at least 1.9× and up to 2.4×. On the x86 cluster, LOCAT shows similar results to those on the ARM cluster.

The rest of this paper is organized as follows. Section 2 introduces the background and motivation. Section 3 presents our LOCAT approach. Section 4 describes the experimental setup. Section 5 provides and analyzes the experimental results. Section 6 describes the related work and Section 7 concludes the paper.

2 BACKGROUND AND MOTIVATION

2.1 Spark SQL Framework

Spark SQL [4] is built on top of Apache Spark [60] to facilitate high-performance structured data processing. Unlike Spark RDD APIs, Spark SQL interfaces provide Spark with more information about the structure of both data and computation being performed [4]. It is therefore widely used in industry [5] such as OLAP [35]. A Spark SQL application typically consists of a number of queries. The Spark SQL framework transforms each query into a DAG which is then split into a collection of stages consisting of a set of parallel tasks. Each task corresponds to a partition computing partial results of an application. Each stage may depend on other stages, called lineage stored in a RDD.

The DAG scheduler of Spark schedules the tasks on several executors to execute in parallel. This parallelism is controlled by several *configuration parameters*. For example, in Yarn [51] mode, the parameter *spark.executor.instances* specifies the number of executors, and *spark.executor.cores* specifies the number of cores used by each executor. The product of these two numbers determines the maximum number of parallel tasks on a Spark SQL cluster.

In summary, the performance of a Spark SQL application is controlled by more than 200 configuration parameters, which can be generally classified into two levels: the Spark SQL configuration parameters (upper level) and the Spark core ones (lower level). The upper level parameters specify the properties of a Spark SQL application. For example, *spark.sql.autoBroadcastJoinThreshold* specifies the maximum size in bytes for a table that is broadcasted to all workers when performing a *join* operation, which significantly affects its performance. The lower level parameters specify fourteen aspects of the Spark core such as *execution parallelism* and *memory management*. Moreover, the upper level configurations may interact

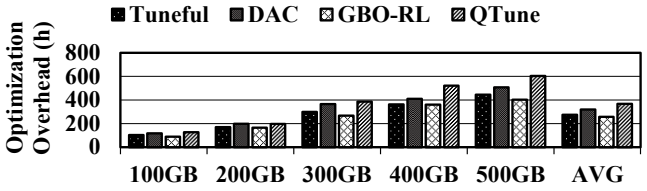

Figure 1: The time used to find the optimal configuration of *TPC – DS* by Tuneful, DAC, GBO-RL, and QTune.

with the lower level ones in a complex way, which makes tuning configurations for a Spark SQL application extremely difficult.

2.2 Bayesian Optimization

Bayesian Optimization (BO) [37] leverages Bayes Theorem to direct an efficient and effective search of a global optimization problem. It minimizes/maximizes an objective function f iteratively through adaptive sampling of the search space. BO has two key components: *surrogate model* and *acquisition function*. The surrogate model is used to model the objective function f and the acquisition function guides the selection of the next sample. BO iteratively fits the surrogate model by using the samples selected by the acquisition function and finally finds the minimal/maximal f.

The surrogate models can be other machine learning models such as Random Forest (RF) and Boosted Regression Trees (BRT) that can well model the non-linear interactions [26]. However, they are weak in theoretical guarantees while GP (Gaussian Process) isn't [32]. Moreover, GP has outstanding features such as supporting for noisy observations and gradient-based methods [43]. We therefore take GP [40, 56] as the surrogate model of BO in this work.

As for acquisition functions, the popular ones are expected improvement (EI) [30], probability of improvement (PI) [25], and GP upper confidence bound (GP-UCB) [40]. Among which, EI is the most widely used one. However, we do not directly use EI. Instead, we leverage the EI with Markov Chain Monte Carlo (EI-MCMC) [45] for better overall performance, which has shown better performance than others across a wide range of test cases [45].

2.3 Motivation

Although the time (e.g., seconds or minutes) used to execute a Spark SQL application is shorter than that (e.g., hours or days) used to optimize its execution, it is still necessary to optimize it because the optimization is a one-shot task while the application is repeatedly executed many times in a long time such as months. Saving a short time in each execution would accumulate a long time, which is a large benefit. To optimize Spark SQL applications, the easiest way is to employ the state-of-the-art (SOTA) approaches. However, we find that these approaches all take a long time (e.g., days or weeks) to find the optimal configuration. Figure 1 shows the time used by four SOTA approaches (Tuneful, GBO-RL, DAC for Spark applications, and QTune for database systems) to find the optimal configuration for *TPC – DS*. We made two observations. For one, the time used by these approaches is at least 89 hours (GBO-RL) when the input data size of *TPC – DS* is 100 GB. Second, the time used by all the approaches is getting significantly longer when the input data size becomes larger.

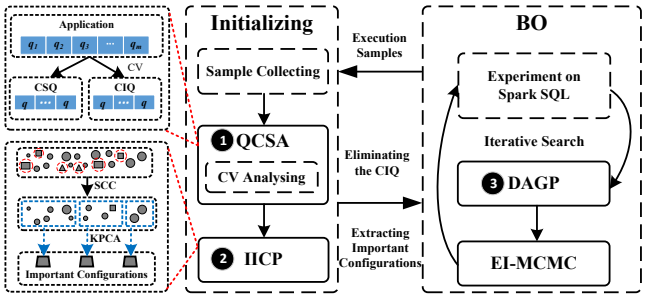

Figure 2: An Overview of LOCAT. BO — Bayesian Optimization. QCSA — Query Configuration Sensitivity Analysis. IICP — Identifying Important Configuration Parameters. DAGP — Data size Aware Gaussian Process. EI-MCMC — Expected Improvement with Markov Chain Monte Carlo.

In industry, the input data size of a Spark SQL application is typically from hundreds of Giga bytes to Tera bytes, even Peta bytes [41]. In such a case, it is very inconvenient, if feasible, to apply the above approaches to find the optimal configurations for Spark SQL applications. For example, we optimize *TPC – DS* with 500 GB of data by using GBO-RL on our ARM cluster, it took 402 hours (16.75 days)! This motivates this work.

3 LOCAT APPROACH

3.1 Overview

LOCAT is a configuration auto-tuning approach that automatically finds the optimal values of configuration parameters for an application running on a given cluster in a short time. It is designed for a common industrial usage: a Spark SQL application repeatedly runs many times with the size of input data changing over time.

Figure 2 shows the block diagram of LOCAT. As can be seen, it consists of three components: query configuration sensitive analysis (QCSA), identifying important configuration parameters (IICP), and data-size aware Gaussian Process (DAGP). QCSA analyzes how the performance (e.g., latency) of each query of a Spark SQL application varies when the configuration parameter values change. If the performance of a query varies significantly when parameter values change, we call it configuration sensitive query. Otherwise, we call it configuration insensitive query. IICP identifies the important parameters for a Spark SQL application to be tuned. DAGP models the performance of a Spark SQL application as a Gaussian Process (GP) of the input data size of the application in addition to the configuration parameters.

When we employ LOCAT to optimize the configurations of a Spark SQL application, we firstly leverage QCSA to identify the configuration insensitive queries of the application and in turn remove these queries. We call the resulted application RQA (reduced query application). Subsequently, we use the component IICP to select the important configuration parameters to tune for the RQA. Finally, the selected configuration parameters and the input data size of the RQA are input to the DAGP which is used as the surrogate model of BO to search for the optimal configuration of the RQA. Note that the optimal configuration of the original Spark SQL application is the same as that of the RQA.

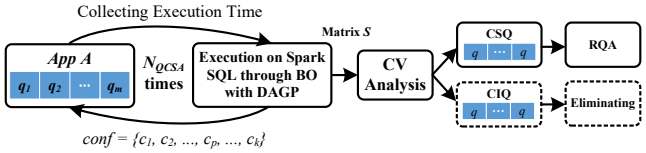

Figure 3: The QCSA diagram. DAGP — Data size Aware Gaussian Process. CV — Coefficient of Variation. RQA — Reduced Query Application. CSQ — Configuration Sensitive Query. CIQ — Configuration Insensitive Query.

3.2 Query Configuration Sensitivity Analysis

As aforementioned, ML-based configuration auto-tuning needs to collect a large number of training samples for an application by running the application on a real cluster the same number of times, which is time-consuming. One possible way to reduce the time is to shorten the execution time of each run. Since a Spark SQL application consists of a number of queries, the execution time of the application would be shortened if some queries can be removed from it. The performance of the removed queries should not be influenced by the value variance of the configuration parameters. Moreover, removing queries should not affect the performance of other queries either. However, we do not know which queries of an application can be removed as collecting training samples for it.

To address this issue, we propose query configuration sensitivity analysis (QCSA) to identify which queries can be removed. Figure 3 shows the block diagram of QCSA. As can be seen, the Spark SQL application *AppA* executes on a given cluster N_{QCSA} times through BO with DAGP, each with a different configuration. A configuration can be represented by a vector as follows.

$$conf = \{c_1, c_2, ..., c_p, ..., c_k\} \tag{1}$$

with c_p the p^{th} configuration parameter value and k the total number of configuration parameters. A random configuration is generated by randomly setting the p^{th} value of $conf$ within the p^{th} parameter's value range and p can be any value between 1 and k. In *AppA*'s each execution, QCSA records each query's execution time. It is represented by $t_{q_{ij}}$ where q_{ij} denotes the i^{th} query of the j^{th} execution of the *AppA*. After the *AppA* executes N_{QCSA} times, we have collected a matrix S denoted as follows

$$S = \{t_{q_{ij}}\}, i = 1, 2, ..., m; j = 1, 2, ..., N_{QCSA} \tag{2}$$

with m the number of queries in *AppA*.

We employ Coefficient of Variation (CV), also known as standard deviation divided by mean, to represent the configuration sensitivity of a query q_i because CV is a standard measure of dispersion of a probability distribution or frequency distribution. In general, higher CV_{q_i} indicates the corresponding query q_i of a Spark SQL application is more sensitive to configuration tuning. To remove some queries from a Spark SQL application when we collect training samples, we need to determine a suitable threshold of CV. However, it is difficult to set a absolute threshold such as 1 for CV because the value ranges of CV for different queries of the same Spark SQL application might be significantly different, let alone different applications. We therefore need a relative way to determine the threshold for CV. It's been proved that classifying CV into *high*, *medium* and *low* is good enough [34, 52] to leverage CV. We therefore equally divide the value range of a CV into three non-overlapped partitions, as shown in equation (3).

$$Width_{CV} = (max(CV_{q_i}) - min(CV_{q_i}))/3 \tag{3}$$

where $Width_{CV}$ is the width of each partition, and $max(CV_{q_i})$ as well as $min(CV_{q_i})$ are the maximum CV and minimum CV of query q_i occurred in the N_{QCSA} executions of *AppA*, respectively. We classify a query with its $CV \in [0, min(CV_{q_i}) + Width_{CV})$ as a configuration insensitive query (CIQ). Otherwise, the query is a configuration sensitive query (CSQ). To collect training samples for *AppA*, we first remove the CIQs and only remain the CSQs, making *AppA* the reduced query application (RQA). Subsequently, we run the RQA a number of times, each with a random configuration. With the same configuration, the execution time of RQA is significantly shorter than that of the original *AppA*. As such, we can collect the same number of training samples as that needed to tune the configuration of *AppA* but with dramatically shorter time.

3.3 Identifying Important Parameters

As mentioned in Section 1, another way to reduce the optimization time needed by ML-based approaches is to reduce the number of training samples by decreasing the number of parameters needing to be tuned. This is because ML-based approaches typically need to construct highly accurate performance models as functions of parameters. For the same accuracy, more training samples are needed to train a performance model if the model takes more parameters as input. We therefore propose to firstly identify the important configuration parameters (IICP) with respect to performance, and subsequently only select the important ones to build performance models. As such, the number of training samples needed to build high accuracy models can be reduced. Figure 4 shows that IICP consists of two stages: sample collection and IICP.

3.3.1 The sample collection stage. It collects the execution times of a small number of executions of *AppA* with a certain input data size, each execution with a random configuration. The execution times and configurations are stored in a matrix S' shown as:

$$S' = \{t_i, conf_i, ds\}, i = 1, 2, ..., N_{IICP} \tag{4}$$

with t_i the execution time of *AppA* with data size ds executed with $conf_i$, and N_{IICP} the number of executions of *AppA*. Smaller N_{IICP} is better because we want to reduce the optimization time.

3.3.2 The IICP stage. This stage extracts the important configuration parameters in terms of performance. There are a lot of approaches such as metric quantification [36], feature selection [11], feature extraction [21] can be used to perform IICP. We do not employ the metric quantification approach used in [36] because it needs a large number of training samples, which is conflict with our goal. We do not use feature selection and extraction directly in this study either. Instead, we employ a *novel hybrid* approach which combines the feature selection and feature extraction.

We therefore employ two steps: *configuration parameter selection* (CPS) and *configuration parameter extraction* (CPE) which seem similar but significantly different. CPS removes the unimportant parameters from the vector $conf$ defined by equation (1) and the remaining ones form a new vector shown in equation (5).

$$r\_conf = \{c_1, c_2, ..., c_i, ..., c_{rk}\} \tag{5}$$

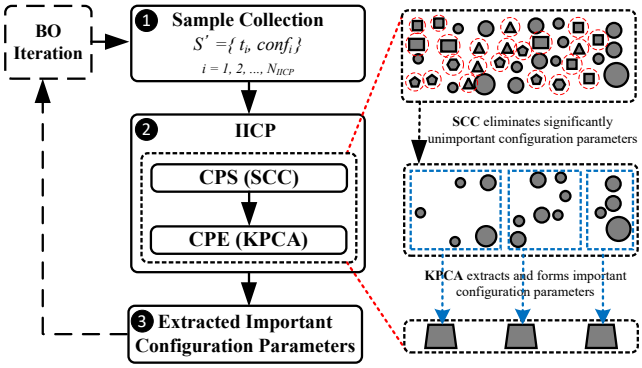

Figure 4: The block diagram of IICP. CPS — Configuration Parameter Selection. SCC — Spearman Correlation Coefficient. CPE — Configuration Parameter Extraction. KPCA — Kernel Principle Component Analysis.

with c_i the i^{th} configuration parameter and rk the number of remaining parameters after CPS is performed. Note that rk is less than k. CPE further extracts important parameters from vector r_conf. Note that these parameters are not the original ones. Instead, they are new parameters which are functions such as linear regressions of the original ones. These small number of new parameters are used to construct the DAGP of BO in this study. After BO converges, we derive the values of the original configuration parameters from the new parameters to optimally configure $AppA$. As such, the time used to search the optimal configuration for $AppA$ can be significantly reduced further.

CPS is implemented by using Spearman Correlation Coefficient (SCC) [61] which is a popular filter approach for feature selection [62]. Compared to Pearson Correlation Coefficient (PCC) [9], SCC is more suitable for IICP because the values of configuration parameters tuned in this study are discrete numerical variables. We calculate the SCC between each configuration parameter c_p and the execution time t_i, and in turn eliminate c_p if the absolute value of its corresponding SCC is less than 0.2 [61], which is a common boundary value of SCC to identify poor correlation [1, 8]. The remaining parameters are stored in r_conf. However, the configuration parameters in r_conf may correlate with each other in a non-linear manner. This indicates that the size of r_conf can be further reduced but it can not be done by using SCC. We therefore design CPE based on the r_conf produced by CPS.

Our CPE is performed by Kernel Principal Component Analysis (KPCA) which is a powerful nonlinear feature extractor [64]. KPCA extends PCA to make it be able to extract non-linear information by leveraging the kernel method. The crucial problem of KPCA is to select a suitable kernel and we select it by experiments. If we use the configuration parameters selected by KPCA with different kernels to configure a Spark SQL application to execute a number of times, the larger standard deviation (SD) of the execution times caused by a kernel indicates that the configuration parameters selected by the kernel are more important to execution time than others.

We evaluate three mainstream kernel methods: Gaussian kernel, perceptron kernel, and polynomial kernel [22] in our experimental environment (Section 4) for two Spark SQL applications: $TPC-DS$ and $TPC-H$. As shown in Figure 5, the SDs of execution times

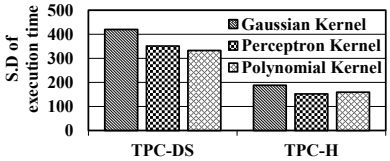

Figure 5: Kernel comparison. S.D — standard deviation.

caused by the Gaussian kernel are the largest for both $TPC-DS$ and $TPC-H$. This indicates that the configuration parameters selected by KPCA with the Gaussian kernel are more important than those selected with other kernels in terms of performance. We therefore choose Gaussian kernel in this study.

3.4 Datasize-Aware Bayesian Optimization

Previous BO-based configuration optimization approaches such as CherryPick [2] can not adapt to the input data size changes of an application. These approaches therefore can not be used online. To address this issue, we design Datasize-Aware Gaussian Process (DAGP) for BO to make it be able to adapt to data size changes.

In detail, we employ GP to model the execution time of a Spark SQL application with a certain configuration as a function of configuration parameters and input data size as follows.

$$t = f(conf, ds) \tag{6}$$

with t the execution time of a Spark SQL application, $conf$ the configuration vector defined by equation (1) used to configure the application, and ds the input data size. As such, LOCAT can adapt to the data size changes of a Spark SQL application during optimization. We express the vector $\{conf, ds\}$ as X_e.

We now elaborate the function f by Gaussian distribution [40] as equation (7).

$$f(X_e) \sim GP(0, K(X_E, X'_E)) \tag{7}$$

where K denotes a covariance matrix. After we run a Spark SQL application with n configurations with different input data sizes, we get a matrix (X_E, T) with n X_e and corresponding t as input training set of GP. The (X_E, T) can be expressed by $(\{X_{e1}, X_{e2}, ..., X_{en}\}, \{t_1, t_2, ..., t_n\})$. X'_E and X_E are two matrices of the same size.

Acquisition function: We use the Expected Improvement (EI) with Markov Chain Monte Carlo (MCMC) hyperparameter marginalization algorithm [45] as BO's acquisition function, which shows better performance than other acquisition functions across wide test cases [45]. BO uses the EI-MCMC to avoid external tuning of GP's hyperparameters, and iteratively selects the next configuration sample with the greatest potential to minimize the execution time of a Spark SQL application.

Start points: LOCAT incrementally builds the GP model, starting with three samples generated by Latin Hypercube Sampling (LHS) [23]. After each execution, the GP model is improved and helps BO pick the next candidate configuration that is estimated to minimize the execution time of a Spark SQL application.

Stop condition: The GP modeling stops after at least 10 iterations and the EI drops below 10%. The goal of setting stop condition is to balance between the exploration of configuration space X_e and the exploitation around the optimal configuration found thus far, which is inspired by CherryPick [2].

Table 1: Experimented Benchmarks and Input Data Sizes.

Benchmark	Input Data Size
TPC-DS	100, 200, 300, 400, 500 (GB)
TPC-H	
HiBench Join	
HiBench Scan	
HiBench Aggregation	

4 EXPERIMENTAL SETUP

4.1 Experimental Clusters and Framework

To evaluate LOCAT, we employ two significantly different clusters: an ARM cluster and an x86 cluster. The ARM cluster consists of four KUNPENG ARM servers. One serves as the master node and the other three servers serve as slave nodes. Each server is equipped with 4 KUNPENG 920 2.60GHz 32-core processors and 512GB PC4 memory. There are in total 512 cores and 2,048 GB memory in the ARM cluster. The x86 cluster consists of eight Xeon severs and one server serves as the master node and the other seven servers are slave nodes. Each x86 server has 2 Intel(R) Xeon(R) Silver 4114 2.20GHz ten-core processors and 64GB PC4 memory. There are in total 160 cores and 512 GB memory in the x86 cluster. The choice of an ARM cluster and an x86 cluster is to evaluate how well LOCAT can adapt to different hardware. Using a four-node and an eight-node cluster is to validate how well LOCAT works in different scales of clusters. On the two clusters, we use Spark 2.4.5 as our experimental framework because of Spark 2.4.5 is more steady and popular in industry compared to other versions.

4.2 Representative Programs

We select the $TPC-DS$ [50], $TPC-H$ [10], and three programs from *HiBench* [27] as representative programs to evaluate LOCAT, as shown in Table 1. $TPC-DS$, containing 104 queries, has been widely used in Spark SQL systems for research and development of optimization techniques [15, 28, 39]. It models complex decision support functions to provide highly comparable, controlled, and repeatable tasks in evaluating the performance of Spark SQL systems [7]. $TPC-H$ benchmark is similar to $TPC-DS$ that simulates a decision support system database environment. We select the $TPC-H$ because it can represent a near-real analysis business with 22 queries only, which is less than $TPC-DS$.

The *HiBench* benchmark suite has been widely used to evaluate the Spark framework and we select three SQL related benchmarks with a single query each in this study: Join, Scan, and Aggregation. 1) Join is a query that typically executes in two phases: Map and Reduce. 2) Scan is a query that consists of only Map operation initiated by the "select" command that splits the input value based on the field delimiter and outputs a record. 3) Aggregation is a query that consists of both Map and Reduce operations. The Map operation ("select" command) first splits the input value by the field delimiter and then outputs the field defined by the Reduce operation("group by" command) as a new key/value pair. In our experiment, we treat these three workloads as three separate benchmarks named *Join*, *Scan*, and *Aggregation*. To evaluate how LOCAT adapts to the dynamic changes of input data size, we employ five different data sizes for our experiments (100GB, 200GB, 300GB, 400GB, and 500GB).

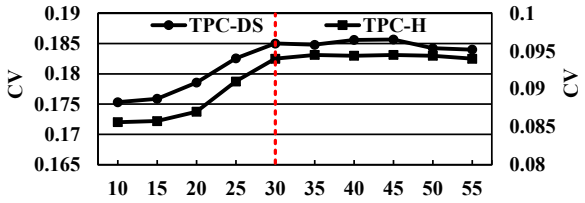

Figure 6: How CV (Coefficient of Variation) changes along with the increasing number of experimental samples for QCSA. The left and right Y axes represent the CVs of $TPC-DS$ and $TPC-H$, respectively.

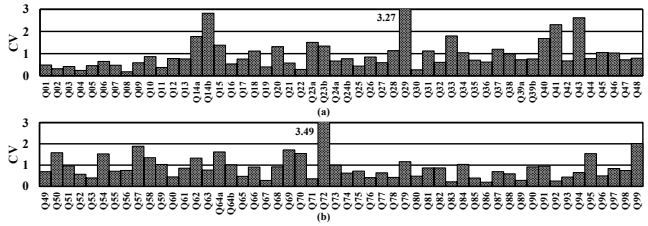

Figure 7: Configuration Sensitivity denoted by CV (coefficient variation) of the $TPC-DS$ queries. Y axis denotes the CVs of queries when configurations are changed.

4.3 Configuration Parameters

The configuration parameters of Spark SQL applications considered in this study are shown in Table 1 of the extended version of this paper [57].

5 RESULTS AND ANALYSIS

In this section, we first determine the numbers of initial experimental samples, N_{QCSA} and N_{IICP}, needed by LOCAT. Subsequently, we present the results and analysis.

5.1 Determining N_{QCSA}

As mentioned in Section 3.2, we need N_{QCSA} of experimental samples to perform the query configuration sensitivity analysis (QCSA). To make the sample collecting time as short as possible, N_{QCSA} should be as small as possible while it should also be large enough to accurately reflect the CV of the Spark SQL queries. We employ experiments to determine a suitable value of N_{QCSA} to satisfy the above requirements. Figure 6 shows that how the CVs for $TPC-DS$ and $TPC-H$ change when we increase the number of experimental samples. As can be seen, when N_{QCSA} increases from 10 to 30, the CV for $TPC-DS$ as well as that for $TPC-H$ keep increasing. When N_{QCSA} is larger than 30, the CVs for both $TPC-DS$ and $TPC-H$ do not increase any more. This indicates that 30 samples are enough for QCSA and we therefore set N_{QCSA} to 30 in this study. Note that we do not collect additional 30 experimental samples before we perform BO with DAGP. Instead, we leverage the samples (executions) performed by the BO iterations.

5.2 QCSA Results and Analysis

After we set N_{QCSA} to 30, we perform QCSA for $TPC-DS$. Figure 7 shows the CVs for the 104 queries of $TPC-DS$. A couple of interesting findings can be observed here. For one, *the CVs for different queries are significantly different*. For example, the CV of query `Q04` is only 0.24 while that of query `Q72` is 3.49. We call the queries

with small CVs configuration insensitive queries (CIQ) while others configuration sensitive queries (CSQ). This indicates that the performance of CIQs such as Q04 does not change much when the configuration changes while that of CSQs such as Q72 does. Second, *long queries are not necessarily sensitive to configuration tuning*. For example, the CV of query Q04 is relatively small (0.24) and its execution time is relatively long (e.g.,80 seconds) while the CV of query Q14b is relatively large (2.8) and its execution time is also relatively long (e.g.,49 seconds). This implies that removing long CIQs such as Q04 can significantly reduce the sample collection time when we collect experimental samples. This also indicates that tuning the configuration of long CSQs such as Q14b can improve performance more than tuning short queries with similar CVs. Why are some queries sensitive to configuration tuning while others are not? The reason is analyzed in Section 5.11 of this paper's extension [57].

Based on these findings, we remove queries by using the CV-based criteria introduced in Section 3.2 for experimental sample collection. For the 104 queries in $TPC-DS$, we remove 81 queries and remain 23 queries when we collect experimental samples. The remaining 23 queries are {Q72, Q29, Q14b, Q43, Q41, Q99, Q57, Q33, Q14a, Q69, Q40, Q64a, Q50, Q21, Q70, Q95, Q54, Q23a, Q23b, Q15, Q58, Q62, Q20}. That is, we only execute 23 queries in each BO iteration with a different configuration during we search the optimal configuration for $TPC-DS$. As such, the time used to collect the experimental samples can be significantly reduced.

5.3 Determining N_{IICP}

To perform IICP, we need N_{IICP} of experimental samples to observe how the performance of a Spark SQL application changes according to the value changes of each configuration parameter. Like N_{QCSA}, the value of N_{IICP} should be as small as possible. On the other hand, N_{IICP} should also be large enough to correctly identify the important parameters with respect to performance.

Again, we employ experiments to determine a suitable value for N_{IICP}. At the first step, we set N_{IICP} to 5 and we therefore run a Spark SQL application five times, each time with a random configuration. The execution times of the five executions and their corresponding configurations are stored in matrix S' defined by equation (4). We then leverage CPS and CPE described in Section 3.3.2 to identify the important configuration parameters with respect to performance. We repeat this step a number of times with each time increasing N_{IICP} by 5. We subsequently observe the number of the identified important configuration parameters. If the number of the parameters keeps constant and parameters remain the same when we perform the IICP with increasing values of N_{IICP}, it indicates that larger N_{IICP} does not help. In our experiments, we tried ten values of N_{IICP} (5, 10, 15, 20, 25, 30, 35, 40, 45, and 50).

Figure 8 shows that the number of the identified important configuration parameters for $TPC-DS$ keeps the same when the value of N_{IICP} is equal to or larger than 20. In addition, the important parameters are also the same when N_{IICP} is larger than 20. We also perform the same experiment for $TPC-H$, Join, Scan, and Aggregation. We find the same phenomenon as shown in Figure 8 and we therefore set N_{IICP} to 20 which is less than the value 30 that we set for N_{QCSA}. Note that we also take the executions in BO iterations as the experimental samples to perform IICP. Therefore,

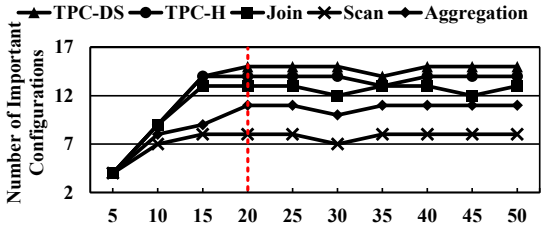

Figure 8: The number variation of identified important parameters along with the increasing number of samples.

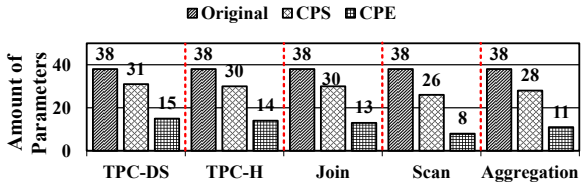

Figure 9: The number of important configuration parameters selected by CPS and CPE.

Table 2: Top 5 important configurations selected by *CPS* with $100GB$, $500GB$, and $1TB$ input data size of $TPC-DS$.

Datasize	100GB	500GB	1TB
Conf (spark.)	sql.shuffle.partitions	sql.shuffle.partitions	sql.shuffle.partitions
	executor.memory	shuffle.compress	shuffle.compress
	executor.cores	executor.memory	executor.memory
	shuffle.compress	executor.instances	executor.instances
	executor.instances	executor.cores	memory.offHeap.size

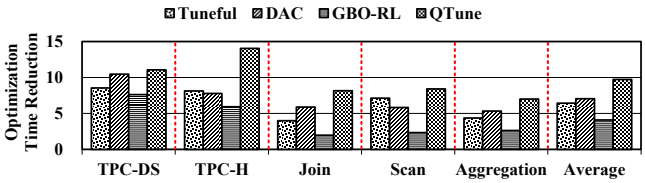

Figure 10: Optimization time comparison between LOCAT and others on the four-node ARM cluster. Y axis denotes the time reduction which is defined by using the optimization time taken by LOCAT to divide those taken by others.

30 experimental samples generated by BO iterations are enough for performing the QCSA and IICP of LOCAT.

Figure 9 shows the number of important configuration parameters identified by CPS and the ones further extracted by CPE. As can be seen, CPS selects about 2/3 of the original 38 configuration parameters as the important configuration parameters for five Spark SQL applications. CPE further extracts about 1/3 of the important configuration parameters selected by CPS. As a result, the number of configuration parameters fed to GP is significantly reduced and in turn the time used to search for the optimal configuration is accordingly dramatically decreased.

5.4 Important Parameter Examples

By using the technique CPS described in Section 3.3, we identify 15 important configuration parameters for the experimented benchmarks. Due to the space limitation, we show the five most important

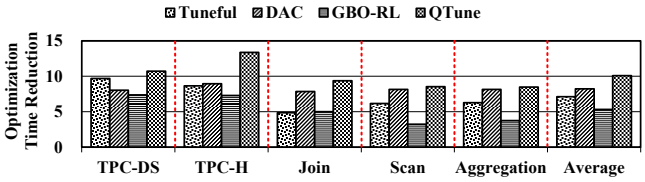

Figure 11: Optimization Time Comparison between LOCAT and others on the eight-node x86 cluster. Y axis denotes the time reduction which is defined by using the optimization time taken by LOCAT to divide those taken by others.

parameters for $TPC-DS$ with three input data sizes in Table 2. A couple of interesting findings can be made here. For one, the most important parameters for the three significantly different input data sizes are all *spark.sql.shuffle.partitions*. This parameter specifies the default number of partitions to use when shuffling data for joins or aggregations. Theoretically, this parameter's value significantly influences the parallelism of shuffle operations, which in turn dramatically impacts the performance of a Spark SQL application.

Second, the three parameters related to the number of executor instances, memory size, and whether compress should be applied on shuffle operations are always in the top five important ones for the three input data sizes, their orders might be different though. The number of executor instances influences the task parallelism; the memory size controls the amount of memory can be used by Spark SQL tasks; and the compress influences the amount of data moved between the servers in the cluster, as well as between the memory and disks. Naturally, these aspects influence the performance of a Spark SQL application significantly.

5.5 Optimization Time

Figure 10 shows the optimization time reduction achieved by LOCAT on the ARM cluster, which is defined by using the optimization time taken by LOCAT to divide those taken by Tuneful, DAC, GBO-RL, and QTune. Note that the input data sizes for the benchmarks are all 300GB. As can be seen, the time taken by LOCAT to achieve the optimal performance of all benchmarks is significantly shorter than those used by other approaches. In detail, the times taken by Tuneful, DAC, GBO-RL, and QTune are 6.4×, 7.0×, 4.1×, and 9.7× of the time used by LOCAT on average, and up to 7.9×, 8.9×, 6.3×, and 11.8×, respectively. Figure 11 shows the results on the x86 cluster. As can be seen, LOCAT reduces the optimization time taken by Tuneful, DAC, GBO-RL, and QTune by factors of 6.4×, 6.3×, 4.0×, and 9.2× on average and up to 9.7×, 8.0×, 7.0×, and 10.3×, respectively. These results indicate two insights. First, LOCAT can indeed significantly reduce the time used by ML approaches to optimize the performance of a wide range of Spark SQL applications. Second, LOCAT can adapt to significantly different hardware as well as different scale of clusters. These insights make LOCAT can be employed in practice.

The optimization time reduction made by LOCAT comes from LOCAT's three innovations. 1) It leverages QCSA to eliminate the executions of configuration-insensitive queries in BO iterations, which significantly reduces the time for executing a Spark SQL application in each BO iteration. As a result, LOCAT significantly reduces the time used for collecting experimental samples. 2) LOCAT accelerates the BO convergence by developing IICP to reduce the dimension of the configuration searching space. 3) LOCAT leverages DAGP to adapt to the data size changes in optimization process, which enables LOCAT to reuse prior results with different input data sizes and in turn reduce the time overhead.

5.6 Speedup

Although LOCAT significantly reduces the optimization time needed by the state-of-the-art (SOTA) approaches, it is still unclear if it can achieve the performance tuned by the SOTA approaches. In this section, we compare the speedups of the program-input pairs tuned by LOCAT over they tuned by Tuneful, DAC, GBO-RL, and QTune. The speedup is defined as

$$speedup = \frac{ET_{sota}}{ET_{locat}} \tag{8}$$

with ET_{locat} and ET_{sota} the execution times of a program-input pair tuned by LOCAT and by a SOTA approach, respectively.

Figure 12 shows the results on the four-node ARM cluster. As can be seen, LOCAT significantly improves the performance of the experimented 25 program-input pairs tuned by other SOTA approaches. In detail, the speedups of the program-input pairs tuned by LOCAT over they tuned by Tuneful, DAC, GBO-RL, and QTune are 2.4×, 2.2×, 2.0×, and 1.9× on average, and up to 3.7×, 3.1×, 2.8×, and 2.4×, respectively. Figure 13 shows the results on the eight-node x86 cluster where LOCAT still significantly outperforms the SOTA approaches in terms of performance. In detail, LOCAT improves the 25 program-input pairs' performance tuned by Tuneful, DAC, GBO-RL, and QTune by factors of 2.8×, 2.6×, 2.3×, and 2.1× on average, and up to 4.8×, 4.7×, 3.7×, and 3.3×, respectively.

A couple of conclusions can be made from these speedups in addition to the optimization time reductions. For one, LOCAT can tune Spark SQL applications with not only higher performance improvements but also in significantly shorter time compared to the SOTA approaches. Second, on significantly different hardware and different scales of clusters, LOCAT can still outperform the SOTA approaches in both performance improvement and optimization time reduction. Third, LOCAT outperforms the SOTA approaches for all different input data sizes of a Spark SQL application, as shown in Figure 12 and Figure 13. Last, LOCAT generally improves the performance more for the larger input data size of a Spark SQL application compared to the SOTA approaches. These benefits make LOCAT more suitable for optimizing future Spark SQL applications because the input data size of them is getting increasingly larger.

Although LOCAT's primary goal is to reduce the optimization time of ML-based tuning approaches for Spark SQL applications, it surprisingly shows performance improvements. This is because LOCAT identifies the important configuration parameters to tune performance. Tuning more configuration parameters does not necessarily result in higher performance. Instead, it may degrade performance because the unimportant parameters may counteract the performance improvements caused by tuning the important ones. We conduct experiments to confirm this. We compare the performance of TPC-DS with input data sizes of 100GB, 200GB, 300GB, 400GB, and 500GB tuned by LOCAT with all the 38 configuration parameters (AP) and with the 15 important parameters (IP) produced by IICP. Figure 14 shows the results. As can be seen, The performance achieved by tuning the 15 important parameters is

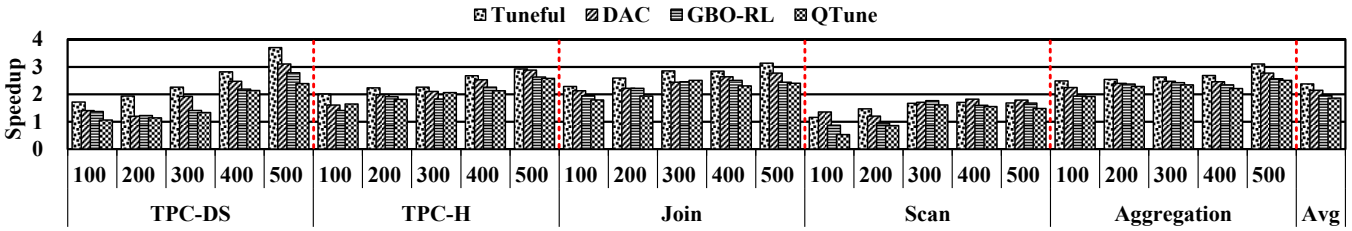

Figure 12: Speedups of the performance tuned by LOCAT over those tuned by Tuneful, DAC, GBO-RL, and QTune on the four-node ARM cluster. The unit of the numbers along with the X axis is GB. The Y axis represents the speedup which is defined by using the execution time of a program-input pair tuned by LOCAT to divide that of it tuned by another approach.

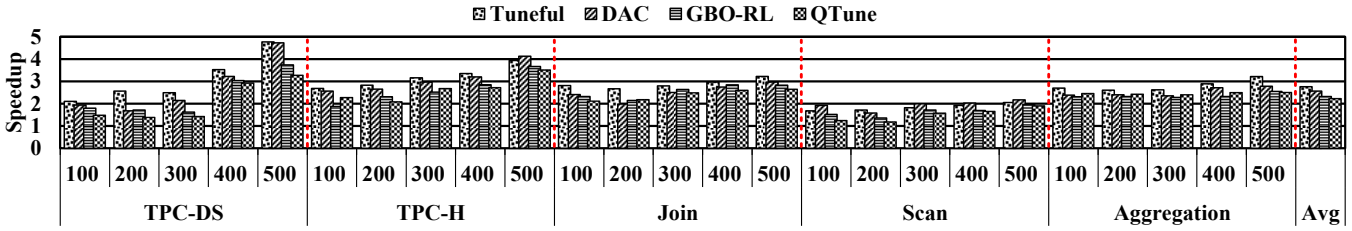

Figure 13: Speedups of the performance tuned by LOCAT over those tuned by other approaches on the eight-node x86 cluster. The unit of the numbers along with the X axis is GB. The Y axis represents the speedup which is defined by using the execution time of a program-input pair tuned by LOCAT to divide that of it tuned by another approach.

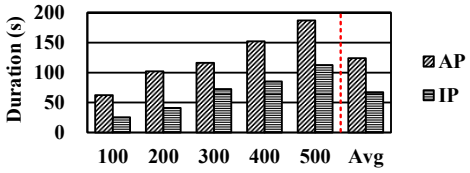

Figure 14: The performance of TPC-DS with input data sizes of 100GB, 200GB, 300GB, 400GB, and 500GB tuned by LOCAT with all parameters (AP) and important parameters (IP).

1.8× higher than that turned by all the 38 ones on average. This confirms that tuning the important parameters results in higher performance than tuning all the ones for Spark SQL applications.

6 RELATED WORK

In this section, we describe the configuration auto-tuning studies related to Spark SQL. A large body of automatic parameter tuning approaches can be applied to Spark SQL, which are divided into six categories [24]: 1) **Rule-based approaches (RBA)** tune performance based on the expert experience, online tutorials [46], or tuning guidebooks [47] which are time-consuming and labour-intensive because using RBA requires a deep understanding of system internals, while LOCAT does not. 2) **Cost modeling approaches' (CMA)** [13, 20, 44, 53, 55, 59] build performance prediction models with analytical model which are not able to be applied to the complex system like Spark SQL and adapt to the input data size changing, while LOCAT is able to. 3) **Simulation-based approaches (SBA)** [3, 17, 18, 31] build performance prediction models based on simulation of optimized system which are not suitable for complex system like Spark SQL. 4) **Experiment-driven approaches (EDA)** [6, 20, 38, 58, 65] find the optimal configuration by executing an application repeatedly with different configuration parameters until converge, which causes high overhead, while LOCAT really takes optimization overhead into consideration and significantly reduces it. 5) **Machine learning approaches (MLA)** [12, 29, 54, 66] build performance prediction models by machine learning algorithms, needing to collect a large number of training samples with high overhead, while LOCAT achieves high optimization performance with significantly low overhead. 6) **Adaptive approaches (AA)** [19, 32, 33, 48] tune the configuration parameter with adaptivity to dynamic runtime status (e.g., input data size changing). AA does not consider the optimization overhead while LOCAT does.

From above, we can find that current approaches still face two problems: First, high overhead of the optimization process. Second, unadaptability of optimal configuration to different input data sizes. Our LOCAT successfully solves these two problems.

7 CONCLUSION

This paper proposes LOCAT, a BO-based approach that efficiently as well as adaptively finds the optimal configurations to achieve high performance for a Spark SQL application on a given cluster. LOCAT innovates three techniques: query configuration sensitive analysis (QCSA), identifying important configuration parameters (IICP), and data size aware Gaussian Process (DAGP). The experiments on two significantly different clusters, a four-node ARM cluster and an eight-node x86 cluster, show that LOCAT can significantly reduce the optimization time of the state-of-the-art approaches and dramatically improve the performance of Spark SQL applications over them.

8 ACKNOWLEDGEMENTS

This work is supported by the Key-Area Research and Development Program of Guangdong Province (Grant No. 2019B010155003), and is partially supported by the Shenzhen Institute of Artificial Intelligence and Robotics for Society (AIRS), Chinese University of Hong Kong, Shenzhen.

REFERENCES

[1] Haldun Akoglu. 2018. User's guide to correlation coefficients. *Turkish Journal of Emergency Medicine* 18, 3 (2018), 91–93. https://doi.org/10.1016/j.tjem.2018.08.001

[2] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. Cherrypick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'17)*. USENIX Association, USA, 469–482. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard

[3] D. Ardagna, E. Barbierato, E. Gianniti, M. Gribaudo, T. B.M. Pinto, A. P.C. da Silva, and J. M. Almeida. 2021. Predicting the performance of big data applications on the cloud. *Journal of Supercomputing* 77, 2 (2021), 1321–1353. https://doi.org/10.1007/s11227-020-03307-w

[4] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797

[5] Lorenzo Baldacci and Matteo Golfarelli. 2019. A Cost Model for SPARK SQL. *IEEE Transactions on Knowledge and Data Engineering* 31, 5 (2019), 819–832. https://doi.org/10.1109/TKDE.2018.2850339

[6] Liang Bao, Xin Liu, and Weizhao Chen. 2018. Learning-based Automatic Parameter Tuning for Big Data Analytics Frameworks. In *2018 IEEE International Conference on Big Data (Big Data)*. 181–190. https://doi.org/10.1109/BigData.2018.8622018

[7] Melyssa Barata, Jorge Bernardino, and Pedro Furtado. 2015. An overview of decision support benchmarks: TPC-DS, TPC-H and SSB. *Advances in Intelligent Systems and Computing* 353 (2015), 619–628. https://doi.org/10.1007/978-3-319-16486-1_61

[8] Helen T. Bhattacharyya, David G. Kleinbaum, and Lawrence L. Kupper. 1979. *Applied Regression Analysis and Other Multivariable Methods*. Vol. 74. Cengage Learning. 732 pages. https://doi.org/10.2307/2287012

[9] Sorana-Daniela Bolboaca and Lorentz Jäntschi. 2006. Pearson versus Spearman, Kendall's tau correlation analysis on structure-activity relationships of biologic active compounds. *Leonardo Journal of Sciences* 5, 9 (2006), 179–200. http://ljs.academicdirect.org/A09/179_200.htm

[10] Peter Boncz, Thomas Neumann, and Orri Erling. 2014. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 8391 LNCS. Springer, 61–76. https://doi.org/10.1007/978-3-319-04936-6_5

[11] Girish Chandrashekar and Ferat Sahin. 2014. A survey on feature selection methods. *Computers & Electrical Engineering* 40, 1 (2014), 16–28. https://doi.org/10.1016/j.compeleceng.2013.11.024 40th-year commemorative issue.

[12] Yuxing Chen, Peter Goetsch, Mohammad A. Hoque, Jiaheng Lu, and Sasu Tarkoma. 2019. d-Simplexed: Adaptive Delaunay Triangulation for Performance Modeling and Prediction on Big Data Analytics. *IEEE Transactions on Big Data* (2019), 1–1. https://doi.org/10.1109/tbdata.2019.2948338

[13] Yuxing Chen, Jiaheng Lu, Chen Chen, Mohammad Hoque, and Sasu Tarkoma. 2019. Cost-effective resource provisioning for spark workloads. In *International Conference on Information and Knowledge Management, Proceedings*. 2477–2480. https://doi.org/10.1145/3357384.3358090

[14] Tatsuhiro Chiba and Tamiya Onodera. 2016. Workload characterization and optimization of TPC-H queries on Apache Spark. In *ISPASS 2016 - International Symposium on Performance Analysis of Systems and Software*. IEEE, 112–121. https://doi.org/10.1109/ISPASS.2016.7482079

[15] Tatsuhiro Chiba, Takeshi Yoshimura, Michihiro Horie, and Hiroshi Horii. 2018. Towards Selecting Best Combination of SQL-on-Hadoop Systems and JVMs. In *IEEE International Conference on Cloud Computing, CLOUD*, Vol. 2018-July. IEEE, 245–252. https://doi.org/10.1109/CLOUD.2018.00038

[16] Alfredo Cuzzocrea. 2015. Data warehousing and OLAP over Big Data: a survey of the state-of-the-art, open problems and future challenges. *International Journal of Business Process Integration and Management* 7, 4 (2015), 372–377. https://doi.org/10.1504/IJBPIM.2015.073665 arXiv:https://www.inderscienceonline.com/doi/pdf/10.1504/IJBPIM.2015.073665

[17] Leandro Batista de Almeida, Eduardo Cunha de Almeida, John Murphy, Robson E. De Grande, and Anthony Ventresque. 2018. BigDataNetSim: A Simulator for Data and Process Placement in Large Big Data Platforms. In *2018 IEEE/ACM 22nd International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. 1–10. https://doi.org/10.1109/DISTRA.2018.8601018

[18] Leandro Batista De Almeida, Damien Magoni, Philip Perry, Eduardo Cunha De Almeida, John Murphy, and Anthony Ventresque. 2019. Multi-Layer-Mesh: A Novel Topology and SDN-Based Path Switching for Big Data Cluster Networks. In *IEEE International Conference on Communications*, Vol. 2019-May. IEEE, 1–7. https://doi.org/10.1109/ICC.2019.8761785

[19] Ayat Fekry, Lucian Carata, Thomas Pasquier, Andrew Rice, and Andy Hopper. 2020. Tuneful: An Online Significance-Aware Configuration Tuner for Big Data Analytics. *arXiv preprint arXiv:2001.08002* (2020). arXiv:2001.08002 http://arxiv.org/abs/2001.08002

[20] Anastasios Gounaris, Georgia Kougka, Ruben Tous, Carlos Tripiana Montes, and Jordi Torres. 2017. Dynamic configuration of partitioning in spark applications. *IEEE Transactions on Parallel and Distributed Systems* 28, 7 (2017), 1891–1904. https://doi.org/10.1109/TPDS.2017.2647939

[21] Isabelle Guyon, Steve Gunn, Masoud Nikravesh, and Lofti A Zadeh. 2008. *Feature extraction: foundations and applications*. Vol. 207. Springer. https://eprints.soton.ac.uk/261922/

[22] Wangli Hao, Jianwu Li, and Xiao Zhang. 2013. Learning KPCA for face recognition. In *Communications in Computer and Information Science*, Vol. 375. Springer, 142–146. https://doi.org/10.1007/978-3-642-39678-6_24

[23] J.C. Helton and F.J. Davis. 2003. Latin hypercube sampling and the propagation of uncertainty in analyses of complex systems. *Reliability Engineering & System Safety* 81, 1 (2003), 23–69. https://doi.org/10.1016/S0951-8320(03)00058-9

[24] Herodotos Herodotou, Yuxing Chen, and Jiaheng Lu. 2020. A Survey on Automatic Parameter Tuning for Big Data Processing Systems. *ACM Comput. Surv.* 53, 2, Article 43 (apr 2020), 37 pages. https://doi.org/10.1145/3381027

[25] Matthew Hoffman, Eric Brochu, Nando de Freitas, et al. 2011. Portfolio Allocation for Bayesian Optimization.. In *UAI*. Citeseer, 327–336. https://dl.acm.org/doi/abs/10.5555/3020548.3020587

[26] Chin-Jung Hsu, Vivek Nair, Vincent W. Freeh, and Tim Menzies. 2018. Arrow: Low-Level Augmented Bayesian Optimization for Finding the Best Cloud VM. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. 660–670. https://doi.org/10.1109/ICDCS.2018.00070

[27] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*. 41–51. https://doi.org/10.1109/ICDEW.2010.5452747

[28] Todor Ivanov and Max-Georg Beer. 2015. Evaluating Hive and Spark SQL with BigBench. *arXiv preprint arXiv:1512.08417* (2015). arXiv:1512.08417 http://arxiv.org/abs/1512.08417

[29] Zhen Jia, Chao Xue, Guancheng Chen, Jianfeng Zhan, Lixin Zhang, Yonghua Lin, and Peter Hofstee. 2016. Auto-tuning Spark big data workloads on POWER8: Prediction-based dynamic SMT threading. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 387–400. https://doi.org/10.1145/2967938.2967957

[30] Donald R. Jones, Matthias Schonlau, and William J. Welch. 1998. Efficient Global Optimization of Expensive Black-Box Functions. *Journal of Global Optimization* 13, 4 (1998), 455–492. https://doi.org/10.1023/A:1008306431147

[31] Soroush Karimian-Aliabadi, Danilo Ardagna, Reza Entezari-Maleki, and Ali Movaghar. 2019. Scalable Performance Modeling and Evaluation of MapReduce Applications. In *High-Performance Computing and Big Data Analysis*, Lucio Grandinetti, Seyedeh Leili Mirtaheri, and Reza Shahbazian (Eds.). Springer International Publishing, Cham, 441–458. https://doi.org/10.1007/978-3-030-33495-6_34

[32] Mayuresh Kunjir and Shivnath Babu. 2020. Black or White? How to Develop an AutoTuner for Memory-Based Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1667–1683. https://doi.org/10.1145/3318464.3380591

[33] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2018. QTune: A QueryAware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment* 12, 12 (2018), 2118–2130. https://doi.org/10.14778/3352063.3352129

[34] Jose Carlos Lorenzo, Lourdes Yabor, Norma Medina, Nicolas Quintana, and Vanessa Wells. 2015. Coefficient of variation can identify the most important effects of experimental treatments. *Notulae Botanicae Horti Agrobotanici Cluj-Napoca* 43, 1 (2015), 287–291. https://www.notulaebotanicae.ro/index.php/nbha/article/view/9881/7790

[35] Yanfei Lv, Huihong He, Yasong Zheng, Zhe Liu, and Hong Zhang. 2015. OLAP query performance tuning in Spark. In *Third International Conference on Cyberspace Technology (CCT 2015)*. 1–5. https://doi.org/10.1049/cp.2015.0832

[36] Yirong Lv, Bin Sun, Qingyi Luo, Jing Wang, Zhibin Yu, and Xuehai Qian. 2018. Counterminer: Mining big performance data from hardware counters. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 613–626. https://doi.org/10.1109/MICRO.2018.00056

[37] Jonas Mockus. 2012. *Bayesian approach to global optimization: theory and applications*. Vol. 37. Springer Science & Business Media. https://doi.org/10.1007/978-94-009-0909-0

[38] Panagiotis Petridis, Anastasios Gounaris, and Jordi Torres. 2017. Spark Parameter Tuning via Trial-and-Error. In *Advances in Big Data*, Plamen Angelov, Yannis Manolopoulos, Lazaros Iliadis, Asim Roy, and Marley Vellasco (Eds.). Springer International Publishing, Cham, 226–237. https://doi.org/10.1007/978-3-319-47898-2_24

[39] Yassine Ramdane, Omar Boussaid, Nadia Kabachi, and Fadila Bentayeb. 2019. Partitioning and Bucketing Techniques to Speed up Query Processing in SparkSQL. In *Proceedings of the International Conference on Parallel and Distributed*

Systems - ICPADS, Vol. 2018-Decem. IEEE, 142–151. https://doi.org/10.1109/PADSW.2018.8644891

[40] Carl Edward Rasmussen. 2004. Gaussian Processes in machine learning. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 3176. Springer, 63–71. https://doi.org/10.1007/978-3-540-28650-9_4

[41] Qayyum Rida. 2020. A Roadmap Towards Big Data Opportunities, Emerging Issues and Hadoop as a Solution. *International Journal of Education and Management Engineering* 10, 4 (2020), 8–17. https://doi.org/10.5815/ijeme.2020.04.02

[42] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on everything. In *Proceedings - International Conference on Data Engineering*, Vol. 2019-April. IEEE, 1802–1813. https://doi.org/10.1109/ICDE.2019.00196

[43] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. 2015. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proc. IEEE* 104, 1 (2015), 148–175. https://doi.org/10.1109/JPROC.2015.2494218

[44] Rekha Singhal and Praveen Singh. 2018. Performance Assurance Model for Applications on SPARK Platform. In *Performance Evaluation and Benchmarking for the Analytics Era*, Raghunath Nambiar and Meikel Poess (Eds.). Springer International Publishing, Cham, 131–146. https://doi.org/10.1007/978-3-319-72401-0_10

[45] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2012/file/05311655a15b75fab86956663e1819cd-Paper.pdf

[46] SparkConf 2022. *Configuration - Spark 3.2.1 Documentation.* Retrieved March 28, 2022 from https://spark.apache.org/docs/latest/configuration.html

[47] SparkTuning 2022. *Tuning - Spark 3.2.1 Documentation.* Retrieved March 28, 2022 from https://spark.apache.org/docs/latest/tuning.html

[48] Pramuditha Suraweera and Antonija Mitrovic. 2002. KERMIT: A Constraint-Based Tutor for Database Modeling. In *Intelligent Tutoring Systems*, Stefano A. Cerri, Guy Gouardères, and Fàbio Paraguaçu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 377–387. https://doi.org/10.1007/3-540-47987-2_41

[49] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: A Warehousing Solution over a Map-Reduce Framework. *Proc. VLDB Endow.* 2, 2 (aug 2009), 1626–1629. https://doi.org/10.14778/1687553.1687609

[50] TPC Benchmark DS. 2020. http://www.tpc.org/tpc_documents_current_versions\/pdf/tpc-ds_v2.3.0.pdf.

[51] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (Santa Clara, California) *(SOCC '13)*. Association for Computing Machinery, New York, NY, USA, Article 5, 16 pages. https://doi.org/10.1145/2523616.2523633

[52] Marcos André Braz Vaz, Paulo Santana Pacheco, Enio Júnior Seidel, and Angela Pellegrin Ansuj. 2017. Classification of the coefficient of variation to variables in beef cattle experiments. *Ciência Rural* 47, 11 (2017). https://doi.org/10.1590/0103-8478cr20160946

[53] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 363–378. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/venkataraman

[54] Guolu Wang, Jungang Xu, and Ben He. 2016. A Novel Method for Tuning Configuration Parameters of Spark Based on Machine Learning. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 586–593. https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0088

[55] Kewen Wang and Mohammad Maifi Hasan Khan. 2015. Performance Prediction for Apache Spark Platform. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. 166–173. https://doi.org/10.1109/HPCC-CSS-ICESS.2015.246

[56] Christopher K.I. Williams and David Barber. 1998. Bayesian classification with gaussian processes. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20, 12 (1998), 1342–1351. https://doi.org/10.1109/34.735807

[57] Jinhan Xin, Kai Hwang, and Zhibin Yu. 2022. LOCAT: Low-Overhead Online Configuration Auto-Tuning of Spark SQL Applications. arXiv:2203.14889 [cs.DC] https://arxiv.org/abs/2203.14889

[58] Zhibin Yu, Zhendong Bei, and Xuehai Qian. 2018. Datasize-Aware High Dimensional Configurations Auto-Tuning of In-Memory Cluster Computing. *SIGPLAN Not.* 53, 2 (mar 2018), 564–577. https://doi.org/10.1145/3296957.3173187

[59] Nikos Zacheilas, Stathis Maroulis, and Vana Kalogeraki. 2017. Dione: Profiling spark applications exploiting graph similarity. In *Proceedings - 2017 IEEE International Conference on Big Data, Big Data 2017*, Vol. 2018-Janua. 389–394. https://doi.org/10.1109/BigData.2017.8257950

[60] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (Boston, MA) *(HotCloud'10)*. USENIX Association, USA, 10. https://dl.acm.org/doi/abs/10.5555/1863103.1863113

[61] Jerrold H. Zar. 2005. *Spearman Rank Correlation.* John Wiley & Sons, Ltd. https://doi.org/10.1002/0470011815.b2a15150

[62] Rizgar Zebari, Adnan Abdulazeez, Diyar Zeebaree, Dilovan Zebari, and Jwan Saeed. 2020. A comprehensive review of dimensionality reduction techniques for feature selection and feature extraction. *Journal of Applied Science and Technology Trends* 1, 2 (2020), 56–70. https://doi.org/10.38094/jastt1224

[63] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 415–432. https://doi.org/10.1145/3299869.3300085

[64] Tonglin Zhang and Baijian Yang. 2016. Big Data Dimension Reduction Using PCA. In *Proceedings - 2016 IEEE International Conference on Smart Cloud, SmartCloud 2016*. IEEE, 152–157. https://doi.org/10.1109/SmartCloud.2016.33

[65] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. BestConfig: Tapping the performance potential of systems via automatic configuration tuning. In *SoCC 2017 - Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 338–350. https://doi.org/10.1145/3127479.3128605 arXiv:1710.03439

[66] Álvaro Brandón Hernández, María S. Perez, Smrati Gupta, and Victor Muntés-Mulero. 2018. Using machine learning to optimize parallelism in big data applications. *Future Generation Computer Systems* 86 (2018), 1076–1092. https://doi.org/10.1016/j.future.2017.07.003