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

연세대학교 컴퓨터과학과 권세인

2022년 10월

SW STAR LAB
Software Technology Advanced Research

과제명: IoT 환경을 위한 고성능 플래시 메모리 스토리지 기반 인메모리 분산 DBMS 연구개발

과제번호: 2017-0-00477

과학기술정보통신부 Ministry of Science and ICT

연세대학교 YONSEI UNIVERSITY

IITP 정보통신기술진흥센터 Institute for Information & communications Technology Promotion

Introduction

It is challenging to tune the configuration parameters of a Spark SQL application for optimal performance because of two reasons.

First, Spark and Spark SQL both have a number (e.g.,> 20) of configuration parameters.

Second, the configuration parameters may intertwine with each other in a complex way with respect to performance.

Recent studies propose to leverage machine learning (ML) to tune the configurations for Spark programs and database Systems.

However, current ML-based approaches have two drawbacks.

First, these approaches take a long time to collect training samples, which is inconvenient in practice.

Second, most ML-based approaches can not adapt to the changes of input data sizes of a Spark SQL application.

LOCAT

we propose **LOCAT** to automatically tune the configurations of a Spark SQL application online.

First : different queries in a Spark SQL application respond to configuration parameter tuning with significantly different sensitivity.

Second : 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.

Third : we propose to **identify the important configuration parameters** of a Spark SQL application and in turn **only tune them in BO (Bayesian Optimization) iterations.**

SPARK SQL FRAMEWORK

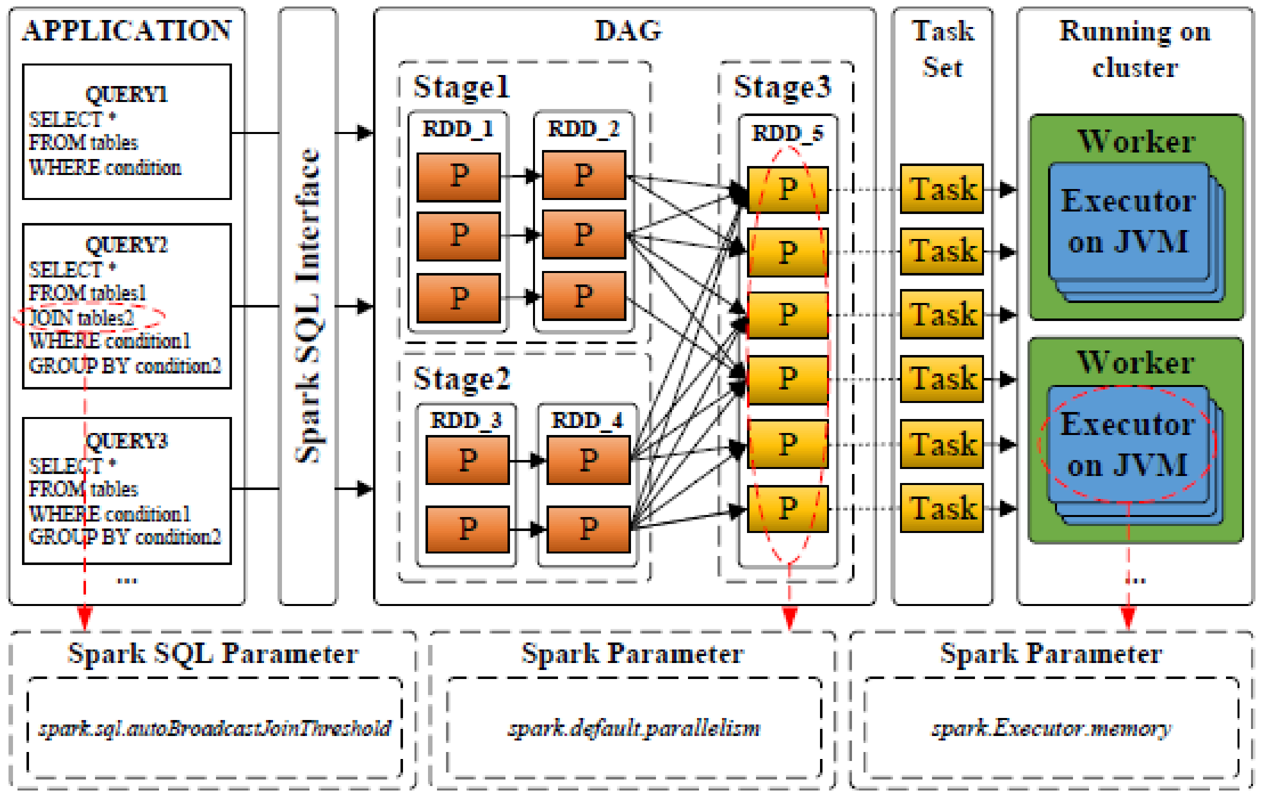


Figure 1: An Overview of the Spark SQL framework.

LOCAT APPROACH

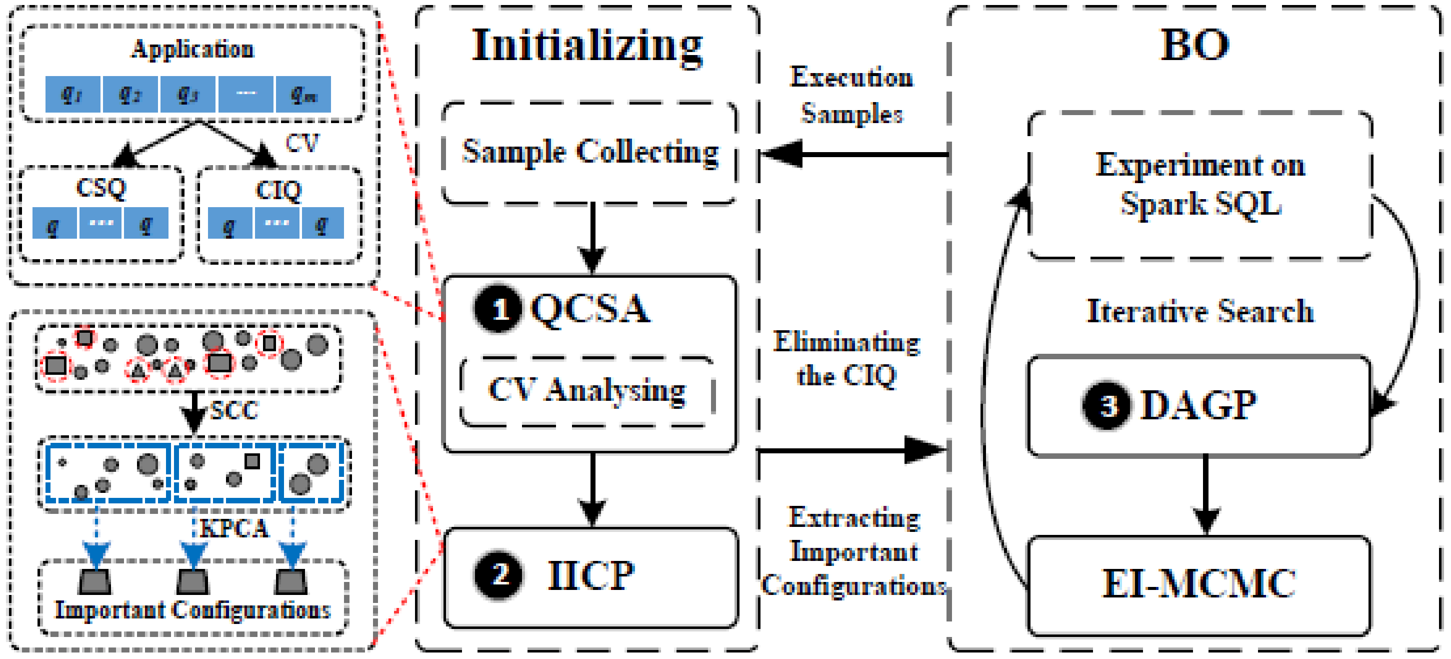
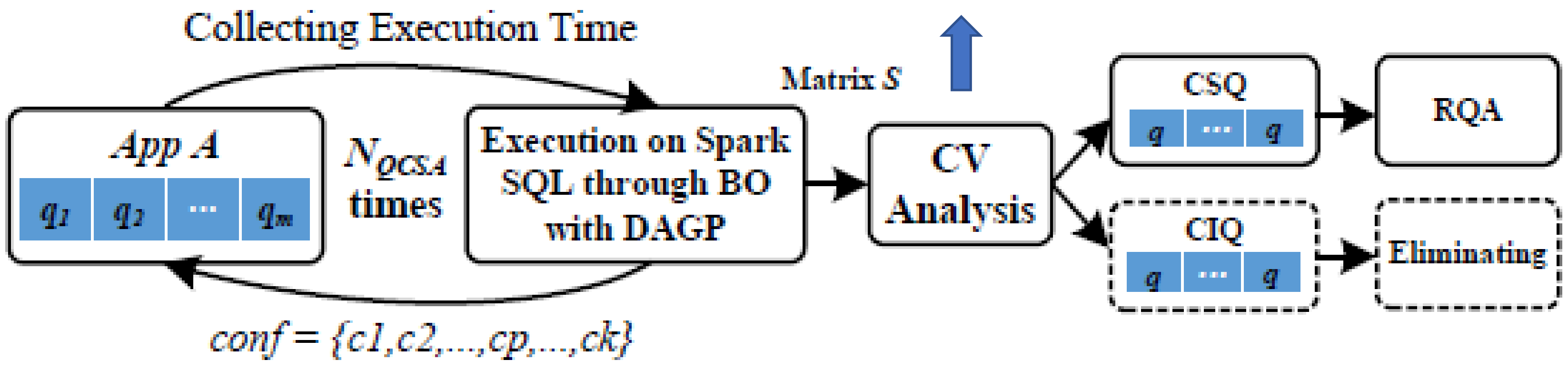


Figure 3: An Overview of LOCAT. BO – Bayesian Optimization. QCSA – Query Configuration Sensitivity Analysis. IICP – Identifying Important Configuration Parameters. DAGP – Data size Aware Gaussian Process. EI-MCMC – Expected Improvement with Markov Chain Monte Carlo.

Query Configuration Sensitivity Analysis

1. Coefficient of Variation (CV)

$$CV_{q_i} = \frac{1}{\bar{t_{q_i}}} \sqrt{\frac{1}{N_{QCSA}} \sum_{j=1}^{N_{QCSA}} (t_{q_{ij}} - \bar{t_{q_i}})^2}$$


Collecting Execution Time
App A
q1 q2 ... qm
N_QCSA times
Execution on Spark SQL through BO with DAGP
Matrix S
CV Analysis
CSQ
q ... q
RQA
CIQ
q ... q
Eliminating
conf = {c1,c2,...,cp,...,ck}

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.

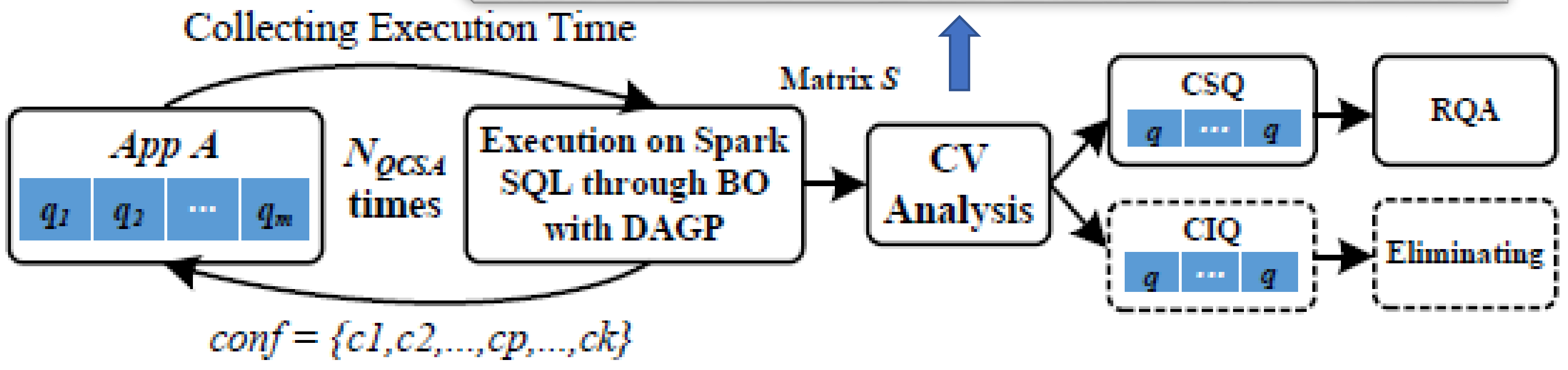
However, we do not know which queries of an application can be removed as collecting training samples for it.

we propose **query configuration sensitivity analysis (QCSA)** to identify which queries can be removed.

Query Configuration Sensitivity Analysis

2. Classifying CV into high, medium and low.

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


Collecting Execution Time
App A
q1
q2
...
qm
N_QCSA times
Execution on Spark SQL through BO with DAGP
conf = {c1,c2,...,cp,...,ck}
Matrix S
CV Analysis
CSQ
q ... q
RQA
CIQ
q ... q
Eliminating

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.

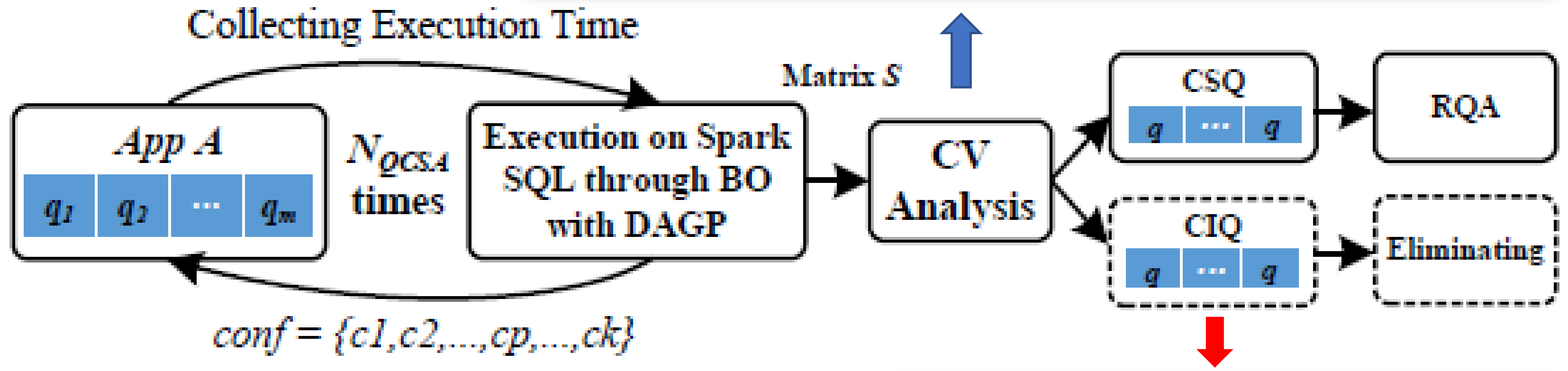
However, we do not know which queries of an application can be removed as collecting training samples for it.

we propose **query configuration sensitivity analysis (QCSA)** to identify which queries can be removed.

Query Configuration Sensitivity Analysis

2. Classifying CV into high, medium and low.

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



$$CV \in [0, min(CV_{q_i}) + Width_{CV})$$

3. Classify a query.

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.

However, we do not know which queries of an application can be removed as collecting training samples for it.

we propose **query configuration sensitivity analysis (QCSA)** to identify which queries can be removed.

Identifying Important Parameters

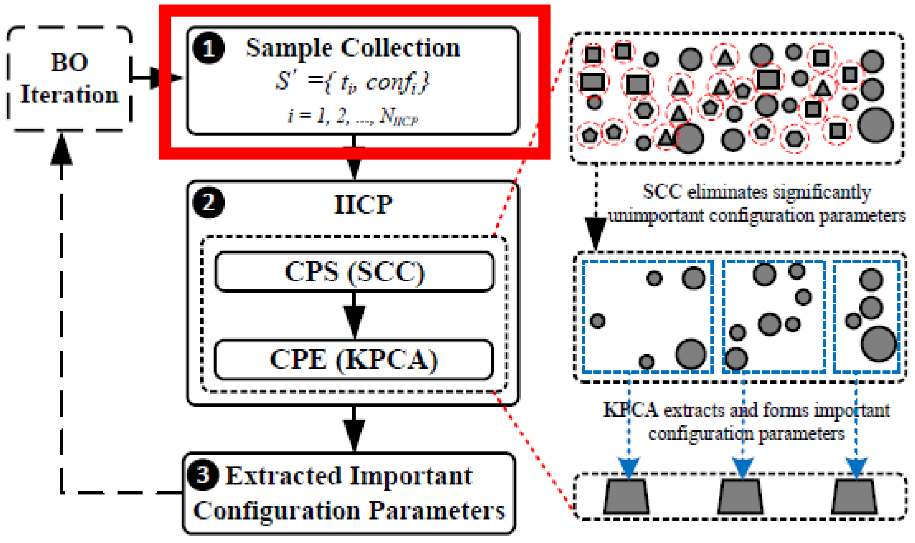

BO Iteration
1 Sample Collection
S' ={ t_i, conf_i}
i = 1, 2, ..., N_IICP
2 IICP
CPS (SCC)
CPE (KPCA)
3 Extracted Important Configuration Parameters
SCC eliminates significantly unimportant configuration parameters
KPCA extracts and forms important configuration parameters

Figure 5: The block diagram of IICP. CPS — Configuration Parameter Selection. SCC — Spearman Correlation Coefficient. CPE — Configuration Parameter Extraction. KPCA — Kernel Principle Component Analysis.

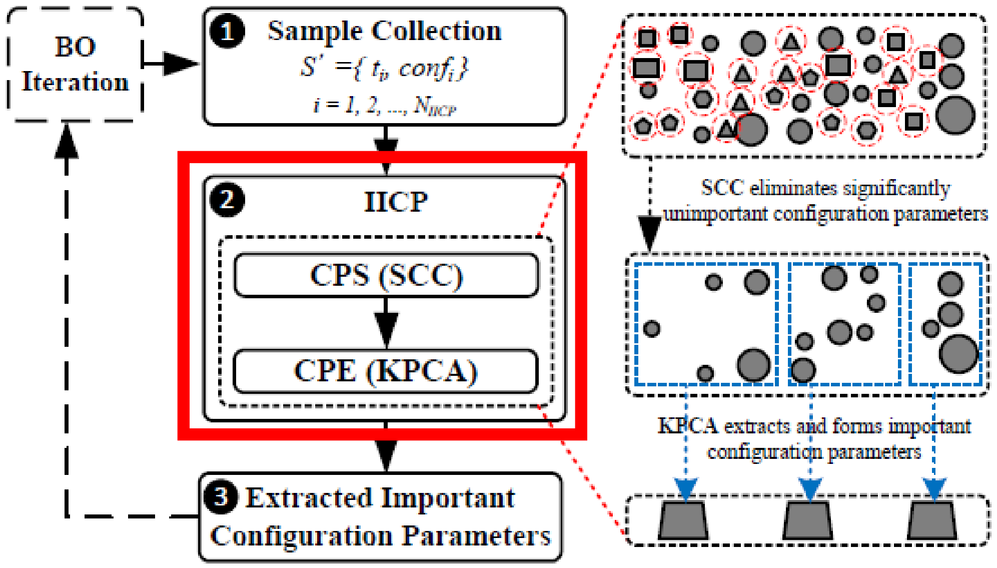
The sample collection stage.

It collects the execution times of a small number of executions of a $AppA$ with a certain input data size, each execution with a random configuration.

$S' = \{ti , confi , ds\}, i = 1, 2, \ldots, \mathrm{N}_{\mathrm{IICP}}$

Smaller $\mathrm{N}_{\mathrm{IICP}}$ is better because we want to reduce the optimization time.

Identifying Important Parameters



BO Iteration
1 Sample Collection
$S' = \{ t_i, conf_i \}$
$i = 1, 2, ..., N_{IICP}$
2 IICP
CPS (SCC)
CPE (KPCA)
3 Extracted Important Configuration Parameters
SCC eliminates significantly unimportant configuration parameters
KPCA extracts and forms important configuration parameters

Figure 5: The block diagram of IICP. CPS – Configuration Parameter Selection. SCC – Spearman Correlation Coefficient. CPE – Configuration Parameter Extraction. KPCA – Kernel Principle Component Analysis.

The IICP stage.

we employ a novel hybrid approach which combines the feature selection and feature extraction.

two steps : configuration parameter **selection (CPS)**
configuration parameter **extraction (CPE)**

Identifying Important Parameters

$$conf = \{c_1, c_2, \ldots, c_p, \ldots, c_k\} \quad \Longrightarrow \quad r\_conf = \{c_1, c_2, \ldots, c_i, \ldots, c_{rk}\}$$

The IICP stage. - configuration parameter **selection (CPS)**

CPS **removes the unimportant parameters** from the vector $conf$ defined by equation and the remaining ones form a new vector shown in equation.

CPS is implemented by using **Spearman Correlation Coefficient (SCC)**.
=> the values of configuration parameters tuned in this study are discrete numerical variables.

Identifying Important Parameters

$conf = \{c_1, c_2, ..., c_p, ..., c_k\}$ $\longrightarrow$ $r\_conf = \{c_1, c_2, ..., c_i, ..., c_{rk}\}$

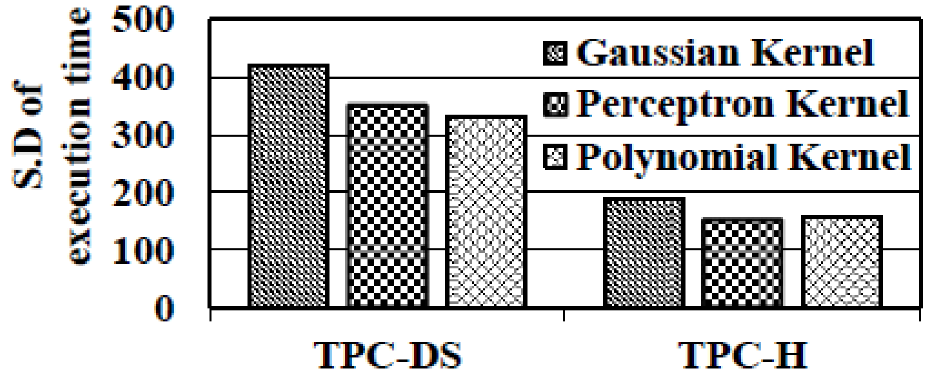


Figure 6: Kernel comparison. S.D – standard deviation.

The IICP stage. - configuration parameter **extraction (CPE)**

CPE further **extracts important parameters from vector r_conf .**

small number of new parameters are used to construct the DAGP of BO in this study.

Our **CPE is performed by Kernel Principal Component Analysis (KPCA)** which is a powerful nonlinear feature extractor.

Gaussian Kernel are more important than other kernels.

Datasize-Aware Gaussian Process

Acquisition Function : Use the Expected Improvement (EI) with Markov Chain Monte Carlo (**MCMC**) hyperparameter marginalization algorithm.

Start points : LOCAT incrementally builds the GP model, starting with three samples generated by Latin Hypercube Sampling (**LHS**) .

Stop condition : The GP modeling stops after at least 10 iterations. The goal of setting stop condition is to balance between the exploration of configuration space and the exploitation around the optimal configuration found.

Summary :

BO starts with the training samples selected by LHS and employs the samples to initialize DAGP.
BO then continuously takes more samples recommended by the DAGP with EI-MCMC until the stop condition is met.
QCSA and IICP are designed to accelerate the optimization process of BO.

Experimental Setup

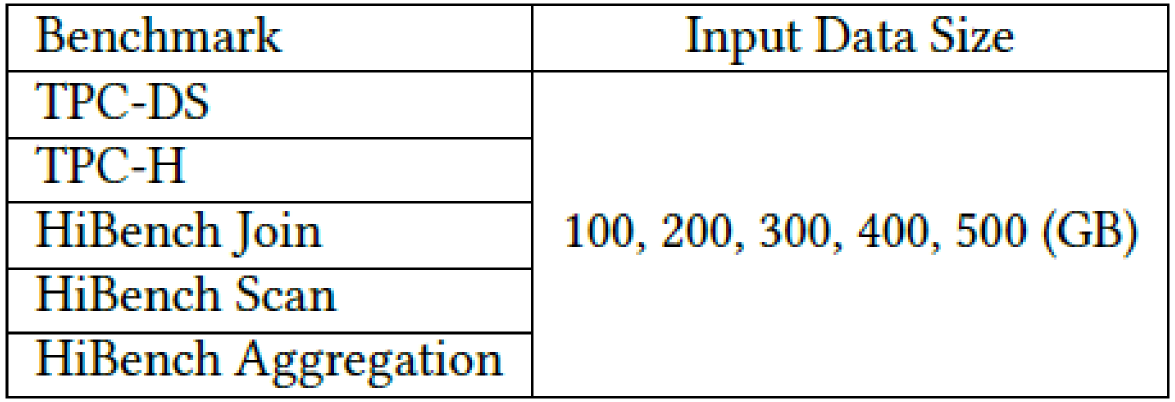
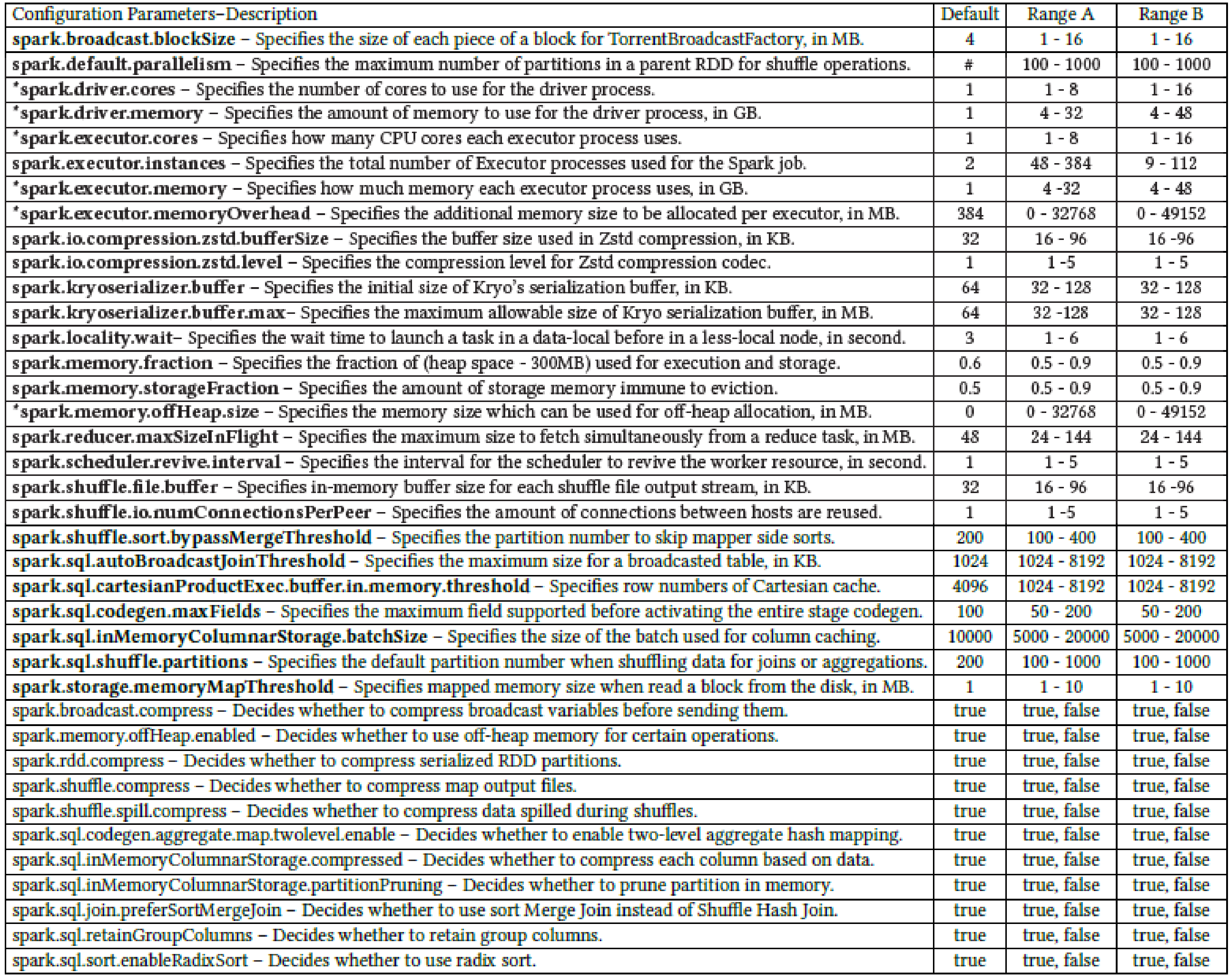


Table 1: Experimented Benchmarks and Input Data Sizes.

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

- Two significantly different clusters: an **ARM cluster and an x86 cluster.**
- Use **Spark 2.4.5**
- To evaluate LOCAT adapts to the changes of input data size, **employ five different data sizes.**

Table 2: Description of Selected Parameters.



Configuration Parameters–Description	Default	Range A	Range B
spark.broadcast.blockSize – Specifies the size of each piece of a block for TorrentBroadcastFactory, in MB.	4	1 - 16	1 - 16
spark.default.parallelism – Specifies the maximum number of partitions in a parent RDD for shuffle operations.	#	100 - 1000	100 - 1000
*__spark.driver.cores__ – Specifies the number of cores to use for the driver process.	1	1 - 8	1 - 16
*__spark.driver.memory__ – Specifies the amount of memory to use for the driver process, in GB.	1	4 - 32	4 - 48
*__spark.executor.cores__ – Specifies how many CPU cores each executor process uses.	1	1 - 8	1 - 16
spark.executor.instances – Specifies the total number of Executor processes used for the Spark job.	2	48 - 384	9 - 112
*__spark.executor.memory__ – Specifies how much memory each executor process uses, in GB.	1	4 -32	4 - 48
*__spark.executor.memoryOverhead__ – Specifies the additional memory size to be allocated per executor, in MB.	384	0 - 32768	0 - 49152
spark.io.compression.zstd.bufferSize – Specifies the buffer size used in Zstd compression, in KB.	32	16 - 96	16 -96
spark.io.compression.zstd.level – Specifies the compression level for Zstd compression codec.	1	1 -5	1 - 5
spark.kryoserializer.buffer – Specifies the initial size of Kryo's serialization buffer, in KB.	64	32 - 128	32 - 128
spark.kryoserializer.buffer.max– Specifies the maximum allowable size of Kryo serialization buffer, in MB.	64	32 -128	32 - 128
spark.locality.wait– Specifies the wait time to launch a task in a data-local before in a less-local node, in second.	3	1 - 6	1 - 6
spark.memory.fraction – Specifies the fraction of (heap space - 300MB) used for execution and storage.	0.6	0.5 - 0.9	0.5 - 0.9
spark.memory.storageFraction – Specifies the amount of storage memory immune to eviction.	0.5	0.5 - 0.9	0.5 - 0.9
*__spark.memory.offHeap.size__ – Specifies the memory size which can be used for off-heap allocation, in MB.	0	0 - 32768	0 - 49152
spark.reducer.maxSizeInFlight – Specifies the maximum size to fetch simultaneously from a reduce task, in MB.	48	24 - 144	24 - 144
spark.scheduler.revive.interval – Specifies the interval for the scheduler to revive the worker resource, in second.	1	1 - 5	1 - 5
spark.shuffle.file.buffer – Specifies in-memory buffer size for each shuffle file output stream, in KB.	32	16 - 96	16 -96
spark.shuffle.io.numConnectionsPerPeer – Specifies the amount of connections between hosts are reused.	1	1 -5	1 - 5
spark.shuffle.sort.bypassMergeThreshold – Specifies the partition number to skip mapper side sorts.	200	100 - 400	100 - 400
spark.sql.autoBroadcastJoinThreshold – Specifies the maximum size for a broadcasted table, in KB.	1024	1024 - 8192	1024 - 8192
spark.sql.cartesianProductExec.buffer.in.memory.threshold – Specifies row numbers of Cartesian cache.	4096	1024 - 8192	1024 - 8192
spark.sql.codegen.maxFields – Specifies the maximum field supported before activating the entire stage codegen.	100	50 - 200	50 - 200
spark.sql.inMemoryColumnarStorage.batchSize – Specifies the size of the batch used for column caching.	10000	5000 - 20000	5000 - 20000
spark.sql.shuffle.partitions – Specifies the default partition number when shuffling data for joins or aggregations.	200	100 - 1000	100 - 1000
spark.storage.memoryMapThreshold – Specifies mapped memory size when read a block from the disk, in MB.	1	1 - 10	1 - 10
spark.broadcast.compress – Decides whether to compress broadcast variables before sending them.	true	true, false	true, false
spark.memory.offHeap.enabled – Decides whether to use off-heap memory for certain operations.	true	true, false	true, false
spark.rdd.compress – Decides whether to compress serialized RDD partitions.	true	true, false	true, false
spark.shuffle.compress – Decides whether to compress map output files.	true	true, false	true, false
spark.shuffle.spill.compress – Decides whether to compress data spilled during shuffles.	true	true, false	true, false
spark.sql.codegen.aggregate.map.twolevel.enable – Decides whether to enable two-level aggregate hash mapping.	true	true, false	true, false
spark.sql.inMemoryColumnarStorage.compressed – Decides whether to compress each column based on data.	true	true, false	true, false
spark.sql.inMemoryColumnarStorage.partitionPruning – Decides whether to prune partition in memory.	true	true, false	true, false
spark.sql.join.preferSortMergeJoin – Decides whether to use sort Merge Join instead of Shuffle Hash Join.	true	true, false	true, false
spark.sql.retainGroupColumns – Decides whether to retain group columns.	true	true, false	true, false
spark.sql.sort.enableRadixSort – Decides whether to use radix sort.	true	true, false	true, false

Results AND Analysis

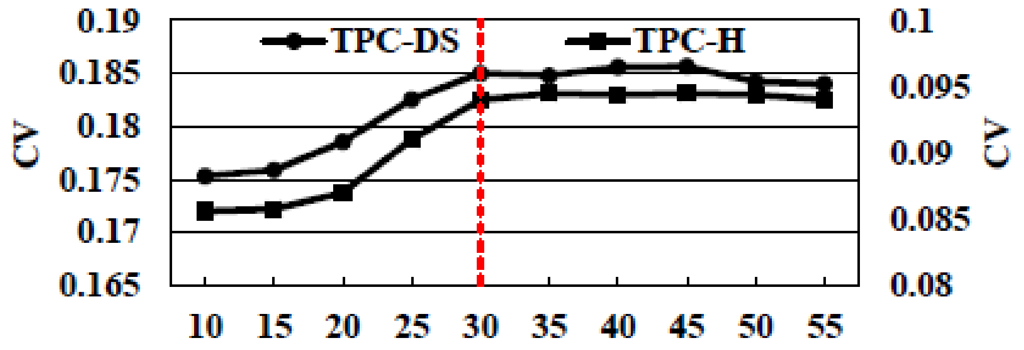


Figure 7: How CV (Coefficient of Variation) changes along with the increasing number of experimental samples for QCSA. The left and right Y axes represent the CVs of $TPC\text{–}DS$ and $TPC - H$, respectively.

- *Determining* $\boldsymbol{N_{QCSA}}$

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.

N_{QCSA} increases from 10 to 30, the CV for TPC-DS as well as that for TPC-H keep increasing.
However, $\boldsymbol{N_{QCSA}}$ **is larger than 30, the CVs are do not increase any longer.**

Indicates that 30 samples are enough for QCSA and more samples do not provide any information for CV besides wasting time.

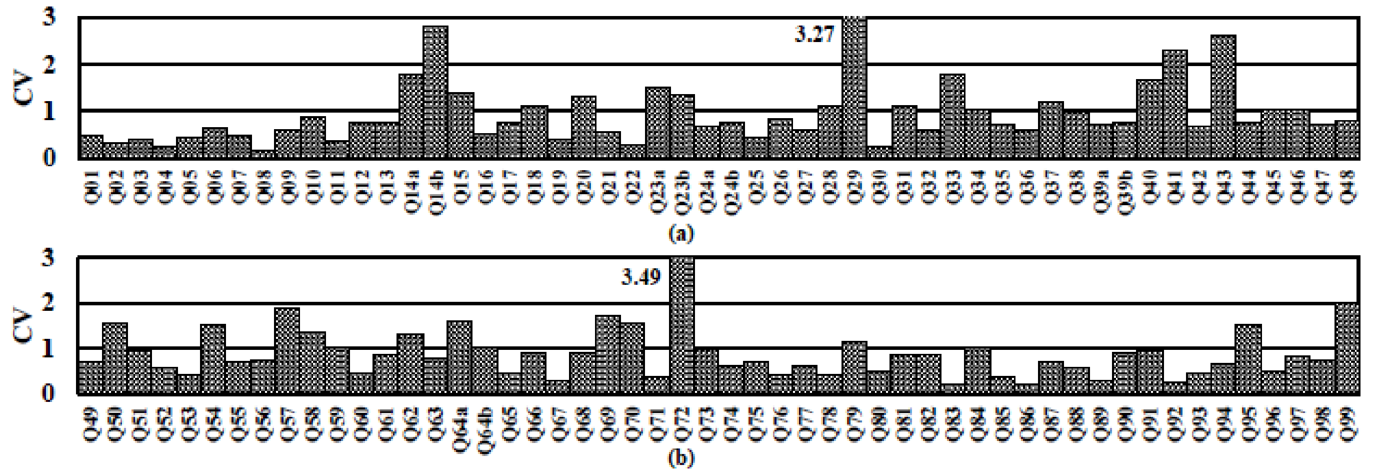


Figure 8: Configuration Sensitivity denoted by CV (coefficient variation) of the $TPC - DS$ queries. Y axis denotes the CVs of queries when configurations are changed.

- *QCSA Results and Analysis*

1. The **CVs for different queries are significantly different.**

2. **Long queries are not necessarily sensitive to configuration tuning.**
    The CV of 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).

Remaining 23 Queries
{Q72, Q29, Q14b, Q43, Q41, Q99, Q57, Q33, Q14a, Q69, Q40, Q64a, Q50, Q21, Q70, Q95, Q54, Q23a, Q23b, Q15, Q58, Q62, Q20}.

Results AND Analysis

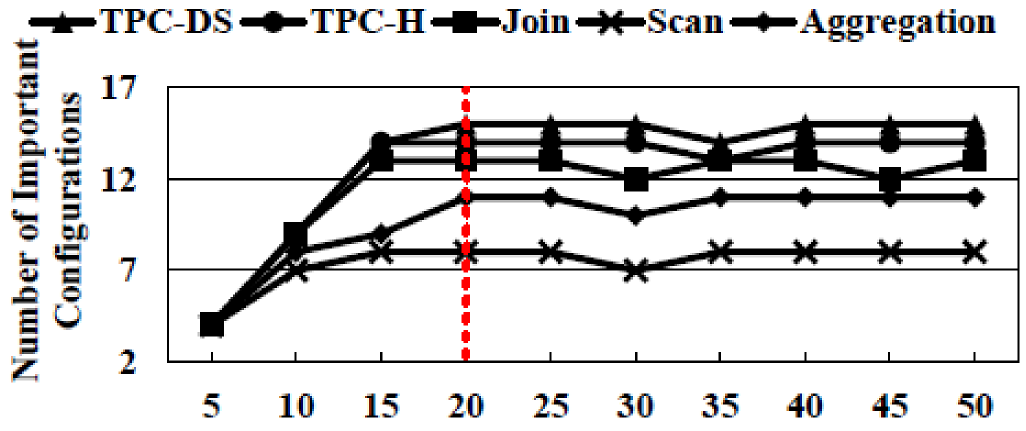


Figure 9: The number variation of identified important parameters along with the increasing number of samples.

- *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.

we set N_{IICP} to 5 and we therefore run a Spark SQL application five times, each time with a random configuration. -> execution time stored in matrix S′ -> leverage CPS and CPE.

Results AND Analysis

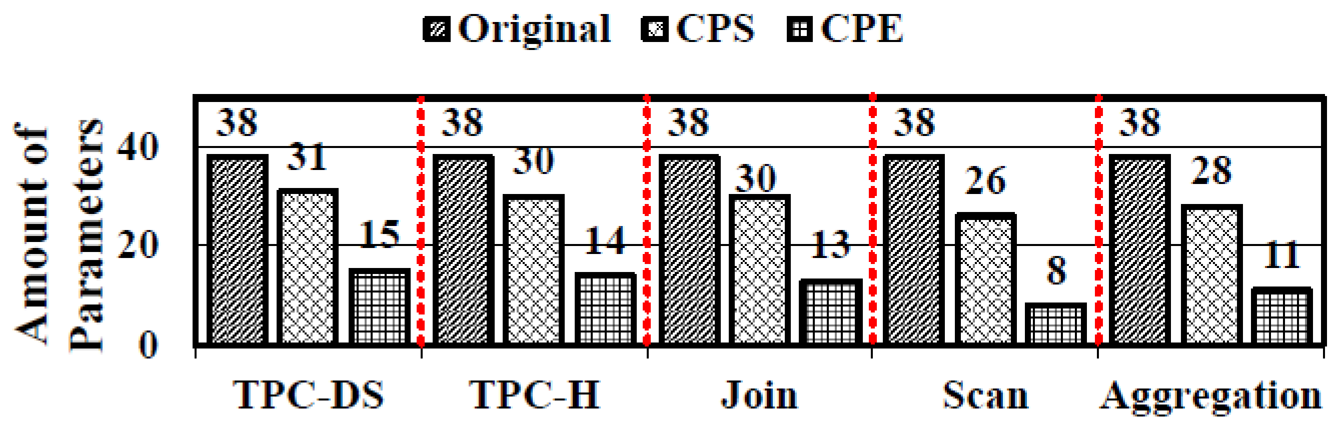


Figure 10: The number of important configuration parameters selected by CPS and CPE.

- *Determining* N_{IICP}

CPS selects about **2/3 of the original 38 configuration parameters.**

CPE further extracts about **1/3 of the important configuration parameters** selected by CPS.

=> time used to search for the optimal configuration is accordingly dramatically decreased.

Results AND Analysis

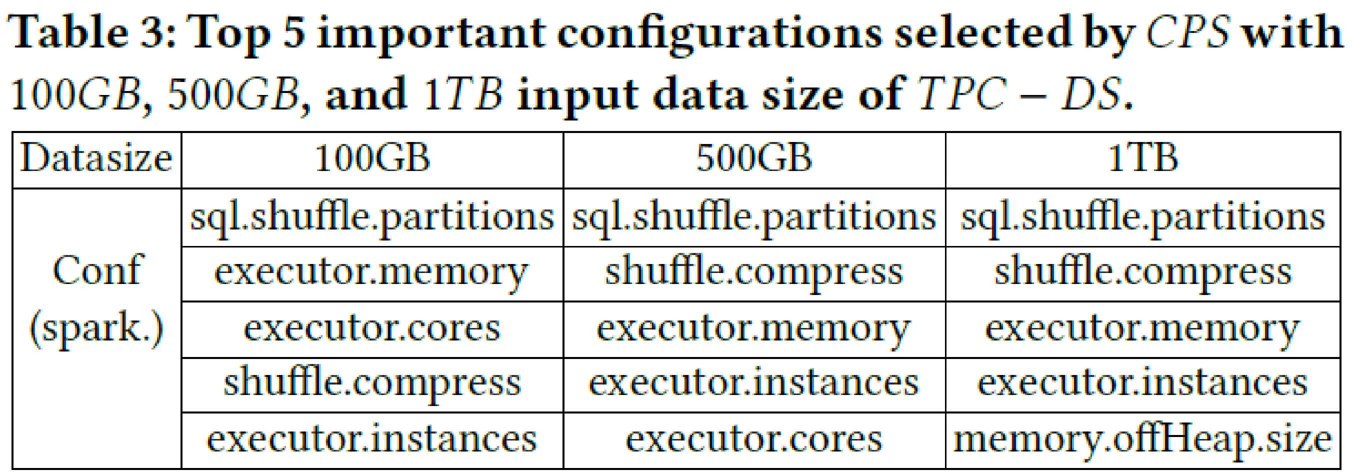


Table 3: Top 5 important configurations selected by *CPS* with 100*GB*, 500*GB*, and 1*TB* input data size of *TPC* – *DS*.

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

- *Important Parameter Examples*

we show the **five most important parameters** for TPC-DS with three input data sizes.

1. the **most important parameters** for the three significantly different input data sizes are all *spark.sql.shuffle.partitions*.

2. The **three parameters related to the number of executor** (instances, memory size, and compress) **always in the top five** important ones for the three input data size.

3. The parameter *Spark.memory.offHeap.size* comes to the fifth most important parameter when the data size increases to 1TB.

Results AND Analysis

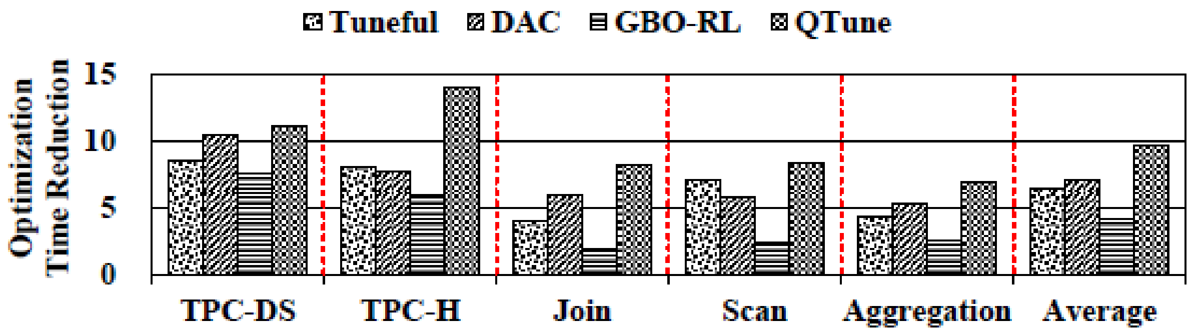


Figure 11: Optimization time comparison between LOCAT and others on the four-node ARM cluster. Y axis denotes the time reduction which is defined by using the optimization time taken by LOCAT to divide those taken by others.

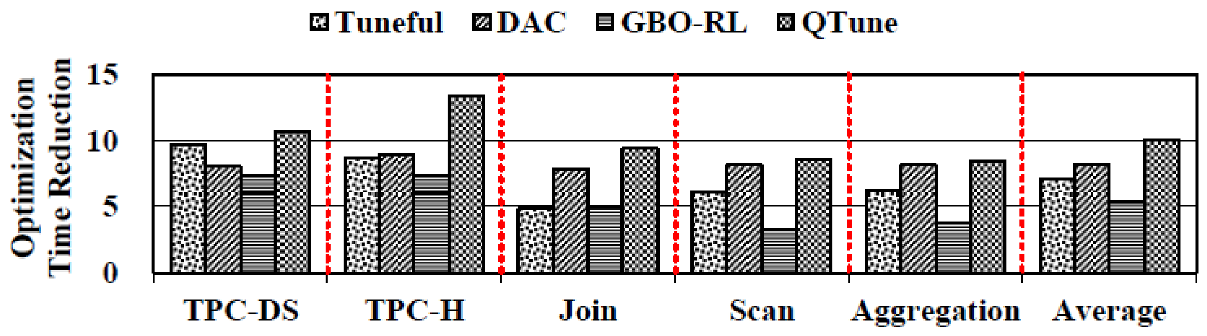


Figure 12: Optimization Time Comparison between LOCAT and others on the eight-node x86 cluster. Y axis denotes the time reduction which is defined by using the optimization time taken by LOCAT to divide those taken by others.

- *Optimization Time*

The optimization time reduction achieved by LOCAT on the ARM cluster.
Note that the input data sizes for the benchmarks are all 300GB.

Others optimization time / Locat optimization time

1. **LOCAT can indeed significantly reduce the time** used by ML approaches to optimize the performance of a wide range of Spark SQL applications.
2. Locat **can adapt to significantly different hardware as well as different scale of clusters**.

Results AND Analysis

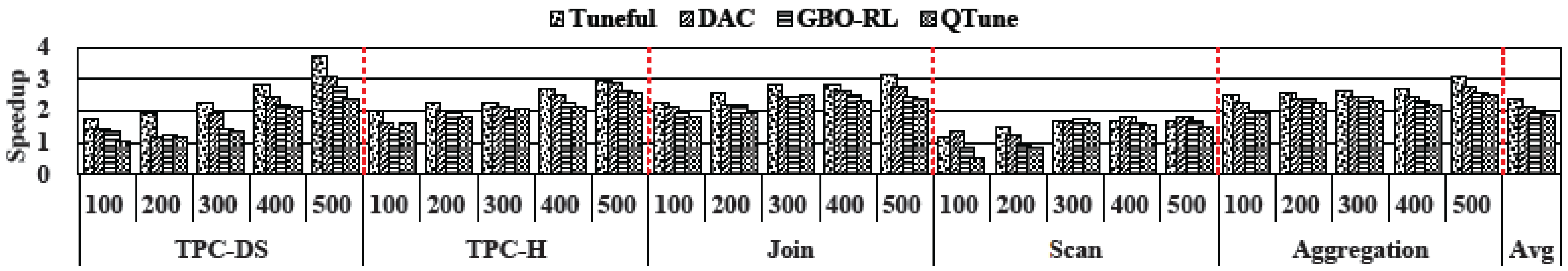


Figure 13: Speedups of the performance tuned by LOCAT over those tuned by Tuneful, DAC, GBO-RL, and QTune on the four-node ARM cluster. The unit of the numbers along with the X axis is GB. The Y axis represents the speedup which is defined by using the execution time of a program-input pair tuned by LOCAT to divide that of it tuned by another approach.

- *Speed Up*

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.

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

Results AND Analysis

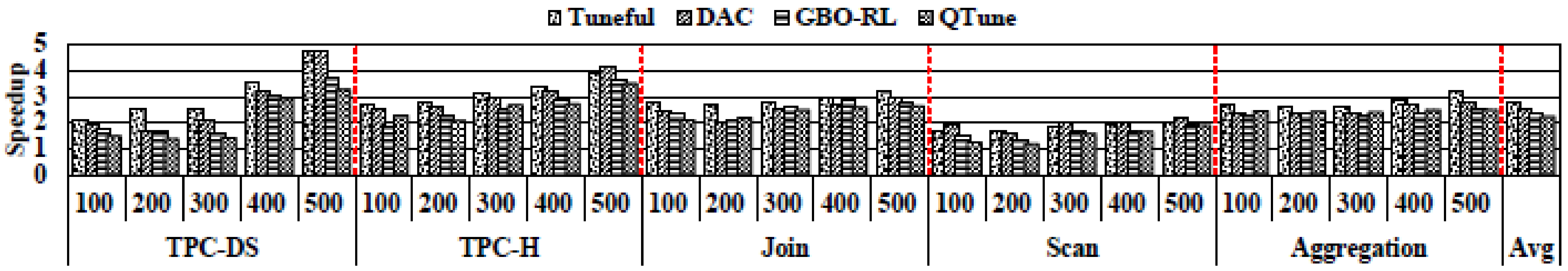


Figure 14: Speedups of the performance tuned by LOCAT over those tuned by other approaches on the eight-node x86 cluster. The unit of the numbers along with the X axis is GB. The Y axis represents the speedup which is defined by using the execution time of a program-input pair tuned by LOCAT to divide that of it tuned by another approach.

1. LOCAT can tune Spark SQL applications with not only higher performance improvements but also in significantly shorter time compared to the SOTA approaches.

2. Different hardware and different scales of clusters, LOCAT can still outperform the SOTA approaches in both performance
   improvement and optimization time reduction.

3. LOCAT outperforms the SOTA approaches for all different input data sizes of a Spark SQL application.

Results AND Analysis

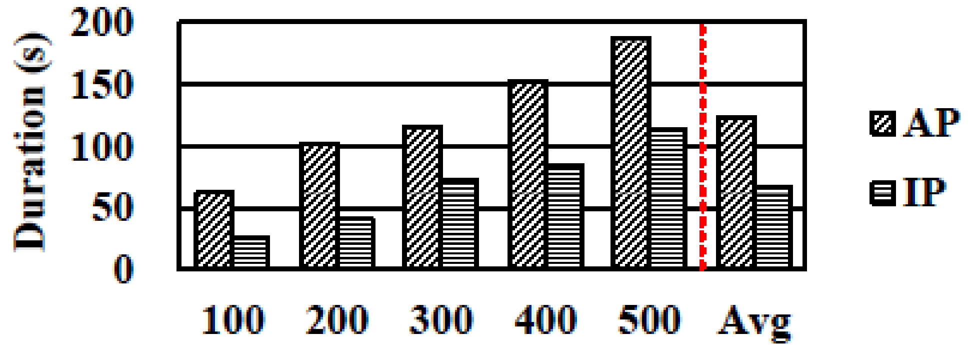


Figure 15: The performance of TPC-DS with input data sizes of 100GB, 200GB, 300GB, 400GB, and 500GB tuned by LOCAT with all parameters (AP) and important parameters (IP).

- *Speed Up*

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.

=> tuning the important configuration parameters results in higher performance than tuning all the configuration parameters for Spark SQL applications.

Results AND Analysis

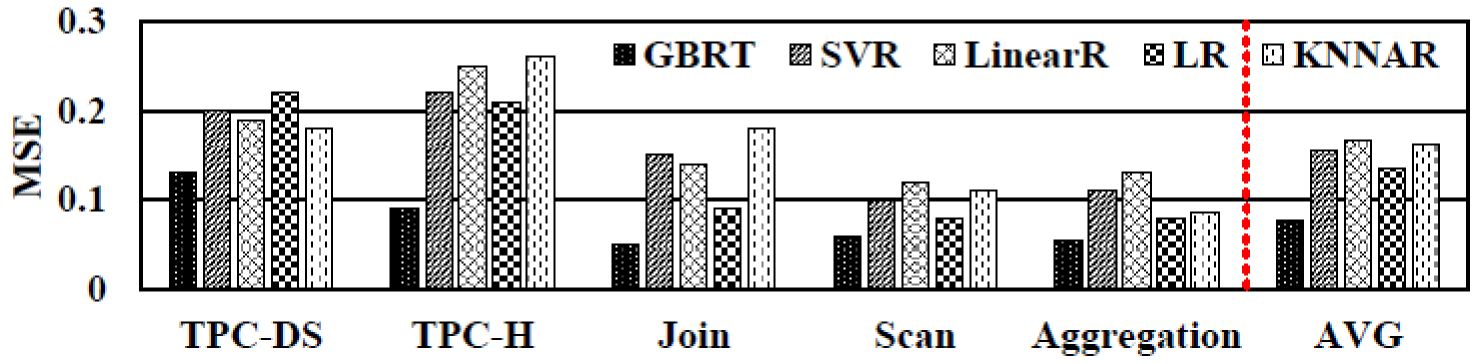


Figure 16: Accuracy of models built by GBRT (Gradient Boosted Regression Tree), SVR (Support Vector Regression), LinearR (Linear Regression), LR (Logistic Regression), and KNNAR (K-Nearest Neighbor Algorithm for Regression).

- *WHY IICP?*

We use several ML algorithms to construct performance models and use the mean squared error (MSE) to measure the model accuracy.

=> The average error of the **GBRT models is the lowest among all models.**

Results AND Analysis

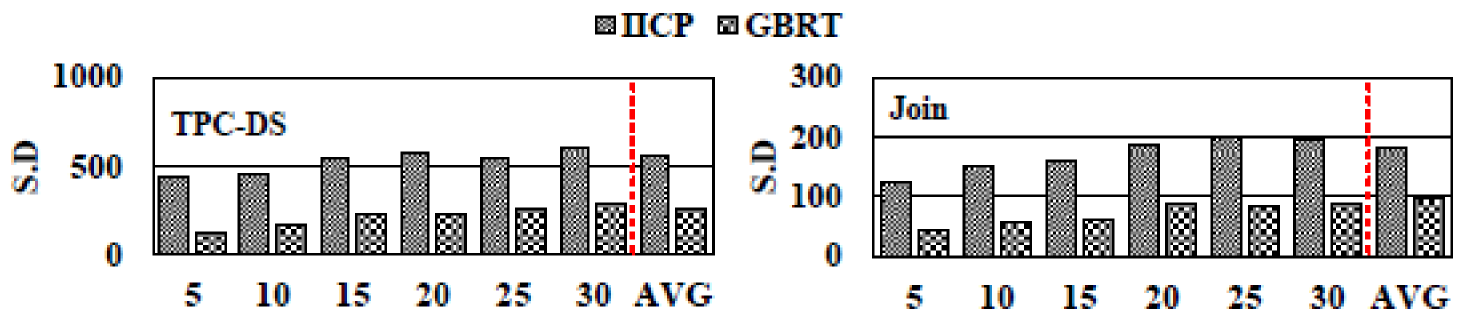


Figure 17: The comparison between IICP and GBRT. The Y axis represents the standard deviation of the execution times of $TPC-DS$ and $Join$ configured by the important parameters identified by IICP or GBRT.

- *WHY IICP?*

Higher SD of execution times indicates that configuration parameters identified by the approach are **more important than one another.**

This indicates that IICP outperforms GBRT for identifying important parameters.

The reason is that GBRT requires a large number of experiment samples to build an accurate model.

Results AND Analysis

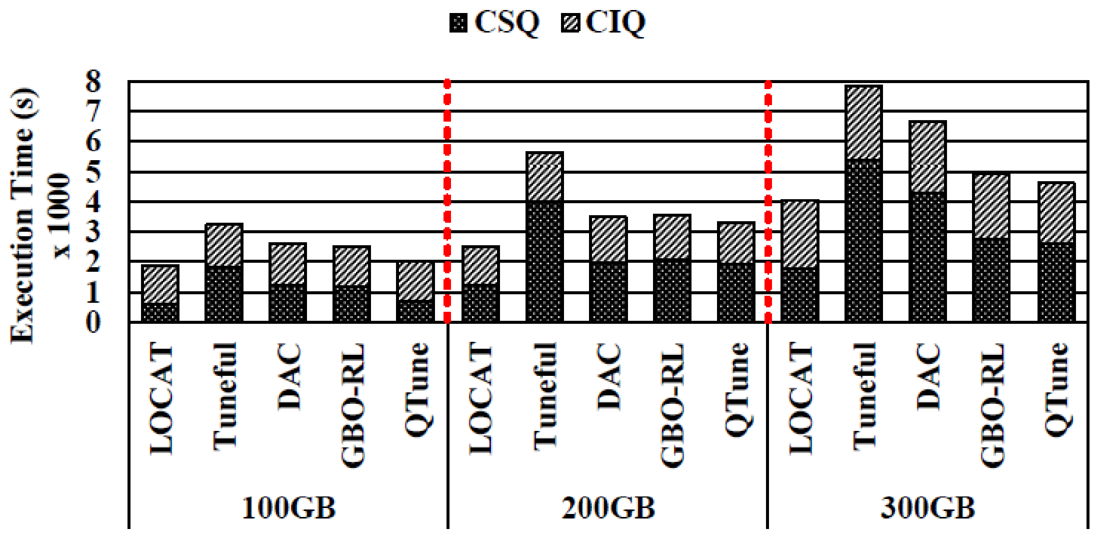


Figure 18: LOCAT Outperforms other Approaches by Significantly Accelerating the Execution of CSQ.

- *Where does the Speedup Come from?*

First, LOCAT, Tuneful, DAC, GBO-RL, and Qtune all reduce the execution time significantly and the performance is higher with larger input data size.

Second, the performance improvement mainly comes from reducing the execution time of CSQ.

Third, LOCAT outperforms other four methods in reducing more executing time of CSQ.

Results AND Analysis

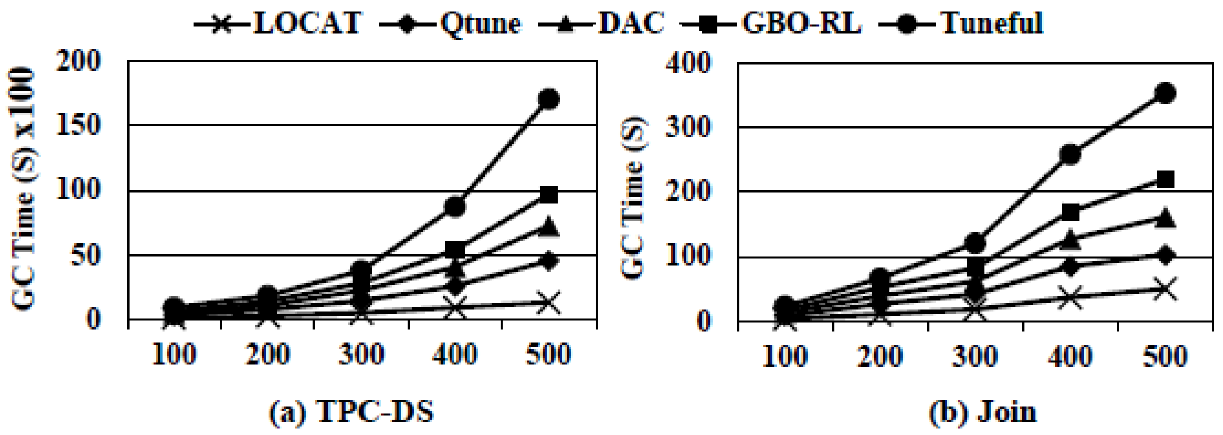


Figure 19: JVM Garbage Collection (GC) Time Comparison.

- *Where does the Speedup Come from?*

Garbage Collection(GC) time comparison for TPC-DS with multiple queries and Join with one query, respectively.

The JVM GC time used by **LOCAT is significantly shorter** than other approaches.

In addition, the GC time used by **LOCAT increases significantly slowly** than other approaches with the increasing of input data size.

Results AND Analysis

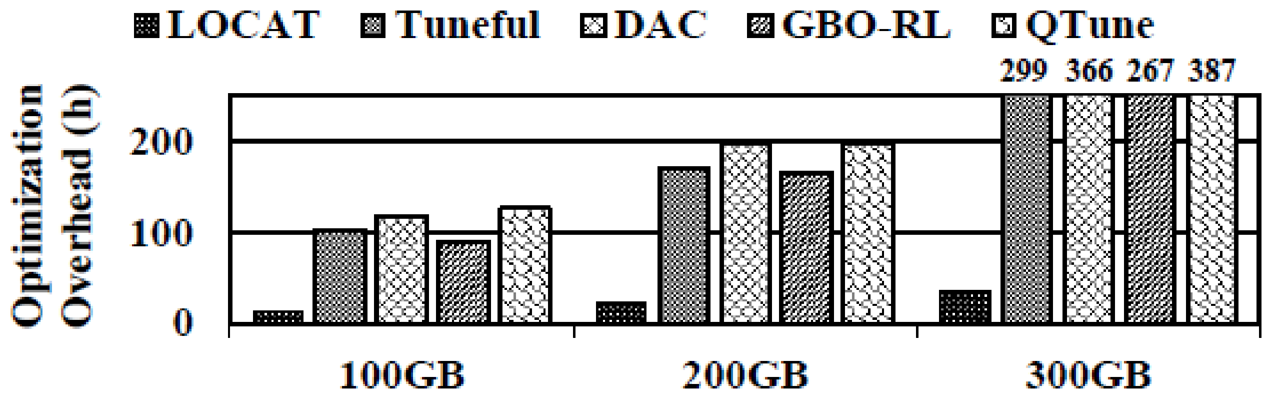


Figure 20: How Tuning Overhead (in hours) Changes when the input data size of an application increases.

- *Tuning Overhead of Increasing Data Size*

When LOCAT and the SOTA approaches are applied to TPC-DS with increasing input data size.

LOCAT incurs **significantly lower optimization overhead** for all **different sizes of input data**.

The reason is that LOCAT adapts to the input data size changes to avoid re-tuning.