ZTune: Model-Assisted Reinforcement Learning for Executor Tuning on Ad Hoc Spark SQL Query

Dejun Kong[1,2], Xiuqi Huang[3], and Xiaofeng Gao[1(✉)]

[1] Shanghai Jiao Tong University, Shanghai 200240, China
kdjkdjkdj99@sjtu.edu.cn, gao-xf@cs.sjtu.edu.cn
[2] University of New South Wales, Kensington, NSW 2052, Australia
[3] State Key Laboratory of CAD&CG, Zhejiang University, Hangzhou 310058, China
huangxiuqi@zju.edu.cn

Abstract. With the development of distributed computing systems, Spark has become widely used in big data parallel processing scenarios. To reduce costs and enhance efficiency, it is essential to make quick and precise adjustments to Spark configuration for each job. Ad hoc Spark SQL job tuning presents unique challenges due to its one-off execution nature, where existing iterative tuning methods for periodic jobs are inapplicable. In this paper, we focus on ad hoc Spark SQL job tuning within Spark executor parameters, which have the most significant influence on processing performance, aiming at improving the quality of service between service providers and clients. We propose a model-assisted reinforcement learning tuning framework, ZTune. ZTune integrates SQL plan representation, a novel query performance model (QPM), and a dual-phase tuning process: prediction and searching. In prediction phase, we combine the constructed model with historical data to better utilize observed patterns and sampling data for more robust performance prediction. In searching phase, we propose a Dual Sampling & Transfer Deep Q-Network to accelerate model training and improve configuration recommendations. Performance evaluation on a large-scale cluster illustrates that ZTune achieves significant performance results compared with baseline methods, including state-of-the-art and industrial approaches, with competitive execution overhead.

Keywords: Spark Tuning · Ad Hoc Query · Reinforcement Learning

1 Introduction

Along with the rapid development of big data processing and analysis, traditional computing on a single machine has encountered a series of problems, such as insufficient computing ability, low computing efficiency, and limited storage capacity [13]. To address these issues, distributed computing clusters are developed. Apache Spark has gradually risen in the past ten years, which is a memory-based distributed computing framework for easier data processing.

H. Jung et al. (Eds.): DASFAA 2026, LNCS 16536, pp. 597–612, 2026.
https://doi.org/10.1007/978-981-92-0366-6_36

Spark configuration is the most important factor influencing the quality of job processing performance. It is composed of multiple parameters involving all aspects of settings, while the importance of different parameters varies [9]. The value range of each parameter contributes to an enormous parameter space. Searching the best configuration group for a specific job can be extremely difficult. What's more, these parameters are to some extent coupled, meaning that modifying one parameter may change the optimal settings for others [8]. In this situation, it is hard to obtain the best configuration by searching each parameter individually. As a result, the configuration searching becomes more complicated. Hence, configuration tuning deserves an in-depth study.

Among all the sections of the Spark framework, the executor plays an important role in job processing, which is responsible for running tasks and storing data for Spark jobs [15]. Recent work implies the importance of the Spark executor parameters in Spark tuning [9]. Executor tuning makes a major contribution to the optimization with a percentage of over 50% [11]. Besides, Spark also supports dynamic allocation of executors, where executors can be added or removed dynamically. Dynamic allocation is also crucial for tuning.

In recent years, there are a number of works on Spark tuning, recommending parameters with various methods. However, there are still some limits as follows.

1. **Lack Attention to Ad Hoc Jobs:** Most recent works concentrate on periodic job tuning [16], which always requires multiple samplings on computing clusters to initialize or assist in optimizing the configuration. Such a tuning mode is suitable for periodic jobs since there are a number of attempts enabling progressive tuning, but not compatible with ad hoc jobs. 25% jobs are ad hoc in industrial scenario [17]. Existing methods rarely consider this.
2. **Limited Optimization Objective:** Most existing works only consider minimizing the processing time when conducting configuration tuning [14]. Some recent works improve the objectives and consider both processing time and resource consumption [11], but the motivation is still simple. Cloud service providers and clients need to balance processing time and resource costs. The processing time also shouldn't be much worse to avoid bad user experiences.
3. **Poor Transparency of Tuning:** A number of recent studies on tuning methods employ machine learning methods to study and predict the relationship between configuration and performance [4]. However, since the processing performance is easily affected by system fluctuation, especially online, it is challenging for machine learning models to distinguish real performance and disturbance. Poor transparency of machine learning cannot guarantee the performance of its mechanism, involving configuration recommendation risks.

In this work, we concentrate on configuration tuning of Spark executors for ad hoc SQL queries. In large-scale workloads, thousands of SQL queries run daily, dominating enterprise applications. Hence, improving the performance of SQL queries contributes to significant benefit improvement. We propose a novel tuning framework in Fig. 1, ZTune, to recommend configurations for ad hoc queries without pre execution, i.e., zero-shot. The whole framework consists of

two phases, prediction phase and searching phase. With the arrival of ad hoc jobs and other history jobs, the tuning system first extracts the SQL plans from the jobs and obtains the representation of the SQL plans. In prediction phase, the framework first trains the regression model of history query performance with history performance data. With the history regression query performance model (rQPM) and SQL plan representation, the framework can train and predict the ad hoc query performance model (pQPM). In searching phase, the framework first selects the most similar history query and trains the Dual Sampling Deep Q-Network model (DS-DQN) with history query performance model and history data. Then the model is transferred to train Transfer DQN model (TDQN) for ad hoc query, recommending the optimal configuration.

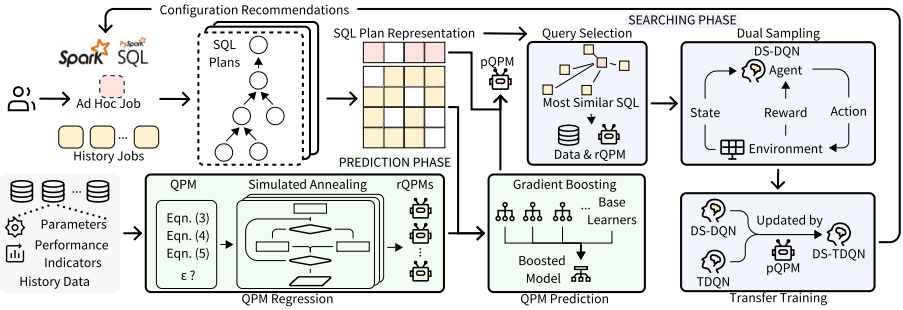


Fig. 1. ZTune: System Overview.

Our contributions are shown as follows to address the limits.

1. Our work fills a critical gap on the study of ad hoc job tuning. We propose a configuration tuning system, ZTune, incorporating multiple objective optimizations to balance processing time and costs with delay penalties.
2. We introduce a query performance model for modeling the relationship between Spark parameters and quality of service, which provides a white box relationship for tuning to enhance the transparency of tuning.
3. We propose a SQL plan representation scheme with three factors: operation number, parallelism, and data size.
4. We design a two-stage performance prediction method with QPM regression and prediction. We also design a Dual Sampling & Transfer Deep Q-Network (DS-TDQN) with query selection, dual sampling, and transfer learning for efficient and better configuration searching.
5. We conduct experiments to evaluate the performance from QPM effectiveness, configuration recommendation, and conditional time-resource balance.

2 Problem Definition

To process Spark jobs, a series of operations need to be completed. In the whole processing scheme, there are four aspects of processing characteristics.

With the whole physical cluster(s) consisting of a number of nodes, there is a group of computing resource features $R = \{r_1, r_2, \cdots, r_m\}$, including cluster node number, cluster core number, cluster memory scale and so on. These features characterize the distributed computing ability of the cluster. As for a Spark job j to be processed, there is a group of job features $J = \{j_1, j_2, \cdots, j_n\}$ to characterize the job, including codes, data size, DAG, physical plan and so on. To process job j, we need to set a series of Spark parameters $S^J = \{s_1^J, s_2^J, \cdots, s_p^J\}$ for processing like spark.executor.cores, spark.executor.memory, spark.dynamicAllocation.maxExecutors. For each Spark parameter $s_i^J, i \in [p]$, there is a value range $s_i^J \in D_i$. With all concerned Spark parameters, there is parameter space $S^J \in \mathbb{S} = D_1 \times D_2 \times \cdots \times D_p$ for parameter settings. With processing environment R, Spark job J and Spark configuration S^J, the job is processed with different performance indicators $U^J = \{u_1^J, u_2^J, \cdots, u_q^J\}$ indicating the performance of the processing procedure, including *app_ duration*, *vcore_ seconds*, *memory_ mb_ seconds* and so on. The definitions of Spark parameters and performance indicators are presented in Table 1.

Table 1. Definitions of Spark parameters and performance indicators

Parameter & Indicator	Definition	Unit
executor.cores	The number of cores to use on each executor.	[1,16]
executor.memory	Amount of memory to use per executor process.	[1,16]
initialExecutors	Dynamic Allocation: Initial number of executors to run.	1
minExecutors	Dynamic Allocation: Lower bound for executor number.	1
maxExecutors	Dynamic Allocation: Upper bound for executor number.	[1,16]
app_duration	The processing time of Spark application.	ms
vcore_seconds	The time integral of applied cores.	core · s
memory_mb_seconds	The time integral of applied memory.	MB · s

Given the above definitions, the goal is to find the best Spark configuration considering the processing performance. The objective function is defined as $\hat{S}^J = \arg\min_{S^J \in \mathbb{S}} F(U^J(R, J, S^J))$, where U^J is a function of R, J and S^J, F is a cost function w.r.t. all $u_i^J, i \in [q]$. Then we aim at finding the best Spark configuration in the parameter space to minimize the cost function F. To evaluate and optimize the overall performance comprehensively considering processing time and resource occupancy, the cost function F is designed as follows. As for a Spark job J, we take the default Spark configuration S_0^J and the corresponding performance U_0^J as a benchmark. Then the performance improvement ratio $r_i^J = \frac{u_i^J}{u_0^J}$ for each $i \in [q]$ is regarded as an important evaluation indicator to measure resource and time optimization. To balance the revenue of both, the cost function $F = r_t^J \times (\sum_{i \in [q]/t} w_i \times r_i^J)^\alpha$ is designed as a weighting function of each resource performance improvement ratio r_i^J and time improvement ratio r_t^J, where α is the weighted factor for the importance between resources and time, and w_i for $i \in [q]/t$ are the weighted factors for the importance among different resources. α and w_i can be determined by users on resources and time.

3 Modelling

3.1 Query Performance Model with Case Study

As for a Spark job, processing time, CPU usage, and memory usage are paid most attention to. Therefore, we focus on the optimization of the three performance indicators *app_ duration*, *vcore_ seconds* and *memory_ mb_ seconds* to balance the processing time and the computing cost. Besides, we select executor.cores, executor.memory, and dynamicAllocation.maxExecutors as tuning parameters to control the executor settings, which contribute most to the processing. We investigate the relationships between these parameters and performance indicators through comprehensive testing on a dedicated cluster using the TPC-DS benchmark. Analysis of the test results revealed consistent and explainable patterns regarding the impact of configuration on performance, as shown in Table 2.

Table 2. The variation pattern of performance indicators and Spark parameters

Indicator	Variation Pattern
app	decreases exponentially as executor.cores increases.
duration	decreases exponentially as maxExecutors increases.
vcore	decreases exponentially as executor.cores increases.
seconds	first decreases then increases or increases as maxExecutors increases.
	decreases exponentially as executor.cores increases.
memory	increases linearly as spark.executor.memory increases.
mb	first decreases then increases or increases as maxExecutors increases.
seconds	Insufficient memory: *app_ duration* increases to a certain extent.
	Insufficient memory: too few executors may cause the job failing.

As there is strong regularity among the three Spark parameters and the three performance indicators, we propose a query performance model (QPM) to express it. QPM consists of three mathematical expressions as follows.

$$app\_duration = \epsilon_t^1 \times cores^{\epsilon_t^2} \times maxExecutors^{\epsilon_t^2} + \epsilon_t^3 \times (\frac{cores}{memory \times maxExecutors})^{\epsilon_t^4} + \epsilon_t^5 \quad (1)$$

$$vcore\_seconds = \epsilon_c^1 \times cores^{\epsilon_c^2} \times maxExecutors^{\epsilon_c^2} \times app\_duration^{\epsilon_c^3} + \epsilon_c^4 \quad (2)$$

$$mem\_seconds = \epsilon_m^1 \times memory^{\epsilon_m^2} \times maxExecutors^{\epsilon_m^2} \times app\_duration + \epsilon_m^3 \quad (3)$$

Equation (1) implies the relationship of *app_ duration* w. r. t. the three indicators. Equation (2) implies the relationship of *vcore_ seconds* w. r. t. the three indicators. Equation (3) implies the relationship of *memory_ mb_ seconds*

w. r. t. the three indicators. All ϵs are hyper-parameters which are determined by the job J and processing environment R. The first term of the three equations represents the major relationship among performance indicators and Spark parameters with a power function. The second term of the *app_duration* equation illustrates the out-of-memory failure (OOM) as shown in Table 2 by setting the *app_duration* to a much larger value so that the OOM situation will not be preferred by ZTune. With QPM, the prediction accuracy of performance is guaranteed by two aspects, equation structure and model fitting.

3.2 SQL Plan Representation

The physical plan of SQL Query is an execution roadmap of a Spark SQL job, which is generated from the SQL code, as shown in Fig. 2. [10] indicate that operation content, SQL structure, and query data are three important aspects. Different operators have specific processing speeds and the number of each operator for a job is the most important information. Second, to represent the plan structure, we define a parallelism factor to describe the parallelism of each operation. First of all, the parallelism factor of the final operator in the directed acyclic graph (DAG) is set to 1. Then, from back to front, once there is more than one branch in the postorder one-for-one operator, the parallelism factor increases by 1. Otherwise, the parallelism factor remains the same as the postorder one. At last, the sizes of all the data tables involved in the physical plan are counted and collected, which decides the processing scale for each operator. With the three parts, we can outline the execution plan framework.

In summary, we denote three groups of encoded representation factors of the physical plan information. The first group includes the operation number factors $\mathcal{X} = \{x_1, x_2, \cdots, x_{op}\}$, in which each factor is the number of one operator and op is the number of operation types. The second group includes the operation parallelism factors $\mathcal{Y} = \{y_1, y_2, \cdots, y_{op}\}$, in which each factor is the sum of all the parallelism factors for one operator. The third group includes one data size factor $\mathcal{Z} = \{z_{size}\}$, where the factor is the sum of all data tables' sizes involved. An example is illustrated in Fig. 2. With the analysis of the submitted SQL query, the representation can be obtained before running the query.

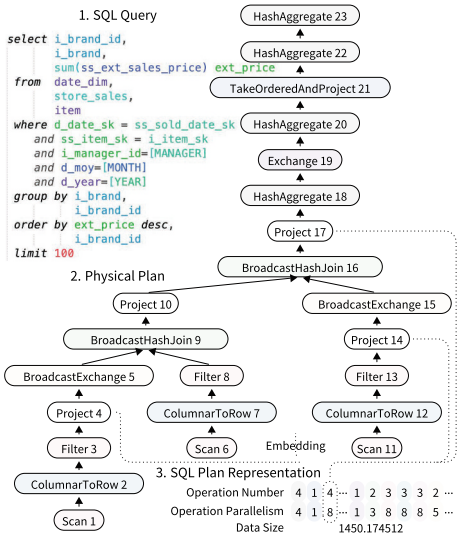


Fig. 2. Physical Plan Representation: Query 55

4 Prediction Phase

The prediction phase consists of a regression module and a prediction module. The regression module is designed for fitting QPM towards historical queries. With regressive QPM (rQPM) and SQL plan representation, the prediction module generates predicted QPM (pQPM) for the target query.

4.1 QPM Regression

In this section, we are going to obtain the QPM of every single historical query to illustrate the characteristics of job processing. For each query, the hyper-parameters in the QPM need to be decided given the history processing data of one query. We employ a Simulated Annealing algorithm to search the best regression for the rQPM as shown in Algorithm 1.

Algorithm 1: Simulated Annealing for Regression

Input: Configuration and performance data for one query
Output: Solution ϵ_* with $* \in \{t, c, m\}$

```
Initialize solution ε_* randomly within configuration space;
Set initial temperature T, cooling rate α, stop criterion T';
while stop criterion not met do
    foreach neighbor ε'_* of ε_* do
        Calculate difference ΔE = MSE(ε'_*) − MSE(ε_*);
        if ΔE < 0 or rand(0,1) < e^{−ΔE/T} then
            Accept ε'_* as the new solution;
    T ← T × α;                                   // cooled temperature
return ε_*;
```

With the history data of query processing, the algorithm starts with an initial random solution ϵ_* and a high temperature T that controls the probability of accepting worse solutions as it explores the configuration space. At each iteration, SA generates a neighboring solution ϵ_* and evaluates its quality by computing an evaluation function. We use mean squared error $\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(* - \hat{*})^2$, where $*$ represents one of the three performance indicators computed by QPM with ϵ_*, $\hat{*}$ represents the true values from history data, and N represents the number of historical processing for one query. If the new solution is better, it is always accepted. If the new solution is worse, it may still be accepted with a probability that decreases as the temperature decreases. This probability is typically determined by the Metropolis criterion. Then the temperature gradually decreases with a rate of α following the iterations until the stopping criterion. With such a design, the algorithm is good at exploring the whole space at first. As temperature approaches the criterion, the algorithm becomes less likely to accept worse solutions and refine the current solution locally.

4.2 QPM Prediction

Following QPM Regression, we study the relationship between the SQL plan and regression model of the history queries to provide a most similar prediction for the target ad hoc query. We employ a gradient boosting algorithm to study rules and predict rQPM for target query as shown in Algorithm 2.

Algorithm 2: Gradient Boosting for Prediction

Input: Representation & QPM regression data for queries
Output: The boosted model: $F(\mathcal{R})$

Initialize dataset $(\mathcal{R}, \epsilon)$, base learner $h_0(x) = 0$;
Initialize number of boosting rounds M, learning rate η;
for $m = 1$ ***to*** M **do**
  Compute the negative gradient: $r_i^m = -\left[\frac{\partial L(\epsilon_i, F(\mathcal{R}_i))}{\partial F(\mathcal{R}_i)}\right]_{F(\mathcal{R})=h_{m-1}(\mathcal{R})}$;
  Fit a weak learner to the negative gradient:
  $h_m(\mathcal{R}) = \arg\min_h \sum_{i=1}^{N} L(\epsilon_i, h_{m-1}(\mathcal{R}_i) + h(\mathcal{R}_i)) + \Omega(h)$;
  Update base learner: $h_m(\mathcal{R}) = h_{m-1}(\mathcal{R}) + \eta \cdot h_m(\mathcal{R})$;
return $F(\mathcal{R}) = \sum_{m=1}^{M} \eta \cdot h_m(\mathcal{R})$;

The algorithm starts with initializing the dataset $(\mathcal{R}, \epsilon)$, where $\mathcal{R}$ represents the embedding vectors of SQL plan, ϵ is the regression target, along with an initialized base learner $h_0(x) = 0$. The number of boosting rounds M and the learning rate η are set at first. For each boosting round m from 1 to M, the negative gradient r_i^m is computed as the direction and magnitude of the current model's prediction error for each sample. A weak learner $h_m(\mathcal{R})$ is then fitted to these negative gradients by minimizing the loss function $L = \frac{1}{N}(r_i^m - (h_{m-1}(\mathcal{R}_i) + h(\mathcal{R}_i)))^2$ and a regularization term $\Omega = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} \beta_j^2$, where T is the node number of the decision tree, β_j is the weight of the j-th leaf node, and γ and λ are the regularization parameters. Then the base learner is updated by adding the scaled predictions of the weak learner. Finally, the boosted model $F(\mathcal{R})$ is produced by summing the contributions of all weak learners. With $F(\mathcal{R})$, the target query is applied to the model, and rQPM is predicted.

5 Searching Phase

In Sect. 4, we have preliminarily established the relationship among Spark configurations and performance indicators by prediction. However, searching for the best configuration of the target query is also a complicated circumstance due to the system fluctuations and accuracy of prediction. We try to utilize the history queries to recognize these cases and recommend the best configuration. In this section, we introduce Dual Sampling & Transfer Deep Q-Network (DS-TDQN) for parameter searching and recommendation. Reinforcement learning is used here for history learning and tuning experience learning.

$$r = \begin{cases} -\frac{app\_dur}{app\_dur_0} \times (w_1 \times \frac{mem\_sec}{mem\_sec_0} + w_2 \times \frac{vcore\_sec}{vcore\_sec_0})^{\alpha}, & app\_dur \leq const \times 2, \\ -100, & app\_dur > const \times 2. \end{cases} \tag{4}$$

5.1 Basic Model

The DS-TDQN model is shown in Fig. 3. The model is first trained on a similar history query towards the target query with history data and rQPM as dual environment. With the pre-trained agent, the model is trained on target query with pQPM to get the recommended Spark configuration.

State. The state space of the reinforcement learning model is a 6-dimension state space, consisting of the three Spark parameters and the three performance indicators as shown in Table 1. The three Spark parameters can be tuned directly by changing the state, while the three performance indicators are decided by the state of the three Spark parameters and environment.

Environment. The environment of reinforcement learning is based on QPM and historical data. The agent samples from the QPM to efficiently obtain the new state, including pQPM and rQPM, as well as historical data.

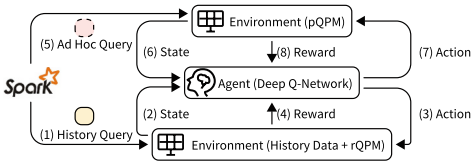


Fig. 3. Dual Sampling & Transfer Deep Q-Network

Action. The action of the reinforcement learning model is to change the value of the Spark parameters. Considering the tuning direction, the action includes increasing, decreasing, or unchanging the Spark parameters with different scales.

Reward. The reward function in Eq. (4) is designed based on the objective function $\hat{S}^J$ in Sect. 2 to satisfy the requirement of both the platform and job provider. In the proposed framework, we set a criterion in which the reward function gets a large penalty if the processing time is beyond two times of the benchmark processing time, making trade-off on resources and time. It ensures a sufficient buffer to accommodate processing uncertainties (e.g., workload fluctuations) and enforces a strict Service Level Agreement adherence.

5.2 Model Optimization

To better adapt the model to Spark configuration searching, we make improvements in three aspects, including query selection, dual sampling and transfer training. Query selection is introduced to select the most similar query for auxiliary configuration tuning. Dual sampling is designed for both considering the regression model and real history data. Transfer learning enables ad hoc query tuning to utilize the history data.

Algorithm 3: DS-TDQN Training

Initialize replay memory D to capacity N & exploration probability ϵ;
Initialize Q-network by transferred weights θ & target Q-network by $\theta^- = \theta$;
for *episode* $= 1$ ***to*** M **do**
  Initialize state s_1;
  for *timestep* $= 1$ ***to*** T **do**
    Select random action a_t or $\arg\max_a Q(s_t, a; \theta)$;
    Execute action a_t, observe reward r_t and state s_{t+1};
    Store transition (s_t, a_t, r_t, s_{t+1}) in D;
    Sample random minibatch of transitions from D;
    Compute target values:

$$y_i = \begin{cases} r_i & \text{if episode terminates at step } i+1 \\ r_i + \gamma \max_{a'} Q(s_{i+1}, a'; \theta^-) & \text{otherwise} \end{cases}$$

    Update θ by minimizing loss: $\mathcal{L}(\theta) = \frac{1}{|B|} \sum_i (Q(s_i, a_i; \theta) - y_i)^2$;
    if *every* C *steps* **then**
      Update target Q-network: $\theta^- = \theta$;

Query Selection. Since we have no definite knowledge on the characteristics of the processing of the ad hoc query, involving a similar history query for pre-training and assistance is an inspiring idea. With the representation vector $\mathcal{R}$ of the SQL plans for all history queries and ad hoc queries, we calculate the Euclidean distance between each history query and ad hoc query, and then find out the minimum one and take the related history query as the similar query. The distance is calculated as $d_{rp} = (v_1 \cdot \sum_{i=1}^{op}(x_i^r - x_i^p)^2 + v_2 \cdot \sum_{j=1}^{op}(y_j^r - y_j^p)^2 + v_3 \cdot (z_{size}^r - z_{size}^p)^2)^{\frac{1}{2}}$, where different vectors can be weighted by v_1, v_2, v_3 to adjust the importance of different parts.

Dual Sampling. As for the history query, there exist errors with the rQPM, while the history data also might not be accurate due to the system fluctuation. To strengthen the sampling quality and obtain more accurate rewards, the model samples both from the rQPM and history data when training on history query. If there is a history job with the same Spark configuration, the model will obtain two groups of results for performance indicators. Then the final performance indicators are obtained by weighted sum as final$(*) = v_1 \cdot *$(from history$) + v_2 \cdot *$(from rQPM), where $*$ represents one of the three performance indicators. If no identical history job, model samples from rQPM only.

Transfer Learning. The transfer learning module is introduced to reduce search costs and improve recommendation quality when target job cannot be directly executed. To accelerate the convergence of training, we transfer the agent pre-trained from history query to initialize the agent for ad hoc query training, so that the experience on similar queries can be utilized for ad hoc query tuning. Specifically, the weights θ_r of the Q-network are transferred to the new training together with randomly initialized weights θ'. The initialized

weights θ are illustrated as $\theta = u_1 \cdot \theta_r + u_2 \cdot \theta'$, where u_1 and u_2 are the weights with $u_1 + u_2 = 1$. This design aims to use the experience appropriately.

5.3 Model Training

With the design of the basic model and optimization, DS-TDQN is trained as shown in Algorithm 3. The training progress includes initialization, action selection, experience replay, and network update, helping to stabilize learning and avoid overfitting to recent experiences. In action selection, the next action is selected randomly with probability ϵ or $\arg\max_a Q$. In network update, the target value y_i is calculated by Bellman function. The network is updated by minimizing the mean square error of Q and y_i. DS-DQN can be pre-trained offline with history, then configuration can be obtained from TDQN for target query.

6 Experiment Setup

In this section, we evaluate the performance of ZTune from three aspects: (1) The ability of QPM model fitting - verify the effectiveness of QPM; (2) The ability of configuration recommendation - test the effectiveness of configuration recommendation towards baseline methods, including traditional, industrial and state-of-the-art methods; (3) The ability of conditional time-resource balance - verify the effectiveness of multi-objective optimization with time criterion.

6.1 Experiment Setting

Environment. We exclusively use 192GB memory and 720 CPU cores of a large-scale Spark cluster in an industrial environment. Tests are repeated five times.

Datasets. We utilize industrial datasets and benchmarks to evaluate our proposed methods, including TPC-DS 500G, which can better simulate massive query scenarios in industrial applications. 99 queries are collected from industrial scenarios, representing different business requirements. We randomly select 9 queries as ad hoc queries for tuning, while the rest are treated as history. Baseline methods are conducted directly interacting with our cluster.

Parameters. A number of hyperparameters used in ZTune are set to definite values in the experiment. α, w_1, w_2 (Reward) are set to $1, 0.5, 0.5$. *const* (Reward) is set to the *app_duration* of the case with moderate resource allocation, (cores, memory, maxExecutors) = $(2, 2, 4)$. T, T', α (SA) and γ, λ (GB) are set to $1, 10^{-9}, 0.1$ / $0.1, 1$. v_1, v_2, v_3 (RL) are set to 1, which means that data size is the most critical, then operation number and parallelism. u_1, u_2 (RL) are set to 0.5 to balance experience transfer and random initialization.

6.2 Baseline Methods

Default Settings. We set a group of default configurations with `cores` = 1, `memory` = 1, and `maxExecutors` = 1 as default input for a job.

Random Search [3]. Random Search recommends random configurations.

Rule-based Method. Rule-based method is based on expert knowledge of Spark. Such rules are usually set to tune several key parameters. In [16], a series of expert rules is introduced for tuning.

CherryPick [1]. The core idea of CherryPick is to utilize Bayesian inference to construct a surrogate model of the objective function and conduct the tuning.

Online Tuning [11]. This method proposes an adaptive multi-objective online tuning method based on Bayesian optimization, effective on periodic jobs. Approximation gradient descent is introduced to accelerate convergence.

ZTune with DQN. DS-TDQN is a variant of DQN under the circumstance of Spark Tuning. To evaluate the improvement of the DS-TDQN model, we test ZTune with the original DQN as the searching phase to analyze the variation.

7 Performance Evaluation

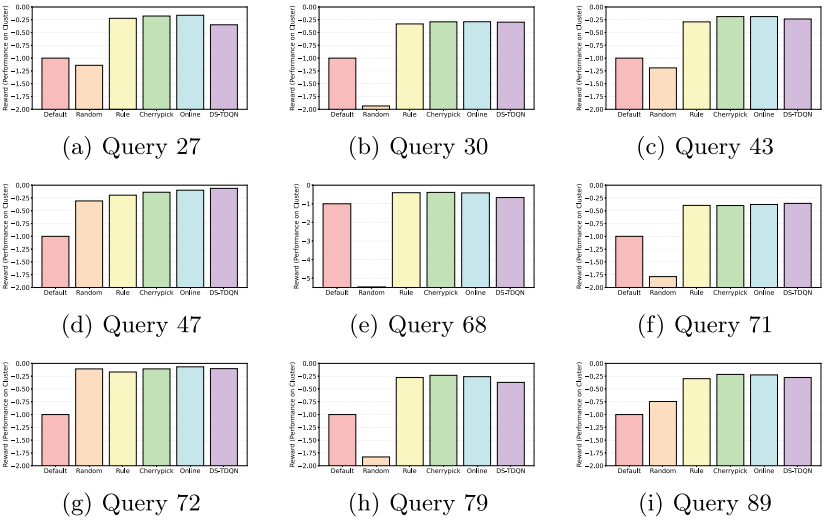
7.1 QPM Effectiveness Analysis

We first evaluate the effectiveness of the designed query performance model by observing the accuracy of model regression. With QPM regression method in Sect. 4.1, we obtain the rQPMs of the selected queries including the 12 ϵ hyper-parameters. Then we calculate the mean square error (MSE) and the average (AVG) for each performance indicator, and the error rates (Error) of the models. The rQPM results of the 9 ad hoc queries are shown in Table 3.

Table 3. Performance Evaluation of QPM: rQPMs of queries (Partial, Error Unit: %)

Query	ϵ_t^1	ϵ_t^2	ϵ_t^3	ϵ_t^4	ϵ_t^5	MSE_t	AVG_t	Error_t	ϵ_c^1	ϵ_c^2	ϵ_c^3	ϵ_c^4	MSE_c	AVG_c	Error_c	ϵ_m^1	ϵ_m^2	ϵ_m^3	MSE_m	AVG_m	Error_m
27	69.78	−1.23	0.00	4.21	16.07	1.87	25.26	7.41	13.87	0.97	0.00	101.96	135.09	495.12	27.29	2174.95	0.81	−15829.67	74448.12	493850.72	15.08
30	50.16	−1.38	0.00	3.49	21.36	2.47	27.22	9.08	7.31	1.08	0.17	107.22	110.45	673.44	16.40	1851.33	0.86	5542.64	88990.67	582460.40	15.28
43	69.18	−1.19	0.00	4.98	14.90	1.83	24.36	7.52	3.60	1.07	0.28	122.82	95.86	464.52	20.64	2399.57	0.79	−26446.83	68712.70	470018.23	14.62
47	387.73	−1.13	3.88	0.55	35.04	5.75	94.62	6.07	4.33	1.24	0.25	577.16	186.98	1567.84	11.93	1685.77	0.89	41095.55	193904.97	1708132.60	11.35
68	31.11	−1.50	0.00	9.89	16.77	1.80	20.09	8.95	1.94	0.81	0.97	−4.27	120.05	455.13	26.38	2090.94	0.82	−6263.97	69473.27	423099.93	16.42
71	43.15	−1.35	17.49	0.00	0.33	1.54	22.96	6.72	13.07	0.95	0.08	79.28	134.29	505.75	26.55	2245.26	0.80	−12261.58	75774.73	467457.61	16.21
72	861.71	−1.13	0.00	5.85	69.33	17.95	198.70	9.03	0.21	1.71	0.54	1457.32	2434.40	3211.10	75.81	1611.87	0.91	140534.64	640742.45	3662199.91	17.50
79	63.21	−1.28	0.00	3.81	17.70	1.98	25.69	7.71	1.65	0.94	0.83	56.44	160.82	537.06	29.94	2106.18	0.82	−10606.34	78291.51	515810.31	15.18
89	59.60	−1.25	0.00	10.00	15.70	1.84	23.46	7.85	2.75	0.68	1.13	−174.13	126.16	463.65	27.21	2234.39	0.81	−20621.52	78009.63	463827.12	16.82

The results in Table 3 imply a good regression outcome on all the queries. The regression error for different queries and performance indicators is restricted to a satisfactory ratio. The regression results of processing time for different queries mostly have an error rate of no more than 10 percent, Based on the results of processing time, the regression results of executor cores and memory also have a good regression error with cores lower than 30 percent and memory lower than 20 percent. This implies that our query performance model is effective for performance regression and prediction. Although there are still regression errors on absolute values, the equation structures can ensure the trend of performance variation so that the prediction phase will not deviate too much from the optimal.



(a) Query 27 (b) Query 30 (c) Query 43

(d) Query 47 (e) Query 68 (f) Query 71

(g) Query 72 (h) Query 79 (i) Query 89

Fig. 4. Tuning Comparison on Rewards: ZTune towards Baseline Methods.

7.2 Configuration Recommendation Analysis

With effective QPM, we conduct comparison experiments on configuration recommendation evaluation, including proposed methods and other baseline methods introduced in Sect. 6.2. As shown in Fig. 4, Default is set as a criterion. Higher reward represents better comprehensive performance on processing time and resource occupancy. Random has large fluctuations on different queries, with mostly poor results due to the randomness. In contrast, Rule has better stability, results, and good iteration numbers on all queries since we modify the parameters of rules based on our recognition and experiment environment towards the SQL queries (QPM). Even more, CherryPick and Online have better rewards as adaptive tuning methods based on Bayesian optimization. Multiple iterations on real clusters enable them to tune more precisely so that a minor advantage towards DS-TDQN is achieved. In comparison, our proposed method DS-TDQN achieves balanced results on reward performance and interaction number. DS-TDQN has better results than Default and Random without interaction, while the results of DS-TDQN approach those of Rule, CherryPick, and Online within one run. DS-TDQN also has better results on queries 47, 71 than other methods.

7.3 Time-Resource Balance Analysis

We further analyze the performance control of processing time to evaluate the realization of the goal. Since the reward discussed above comes from weighted processing time and resources, it means that better resource saving contributes to worse time improvement. The processing time improvement ratio is the ratio of the processing time of different methods towards the processing time of default

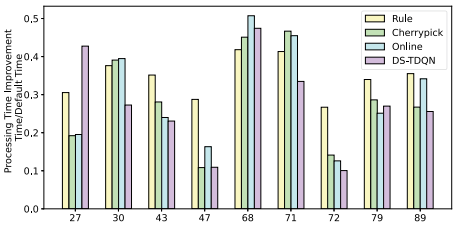


Fig. 5. The Improvement of Processing Time: The comparison of the processing time improvement ratios among Rule-based Method, CherryPick, Online Tuning, and ZTune (DS-TDQN).

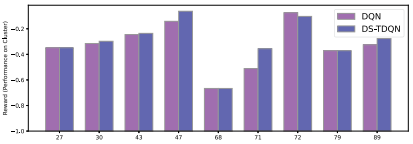


Fig. 6. Performance: DS-TDQN vs DQN

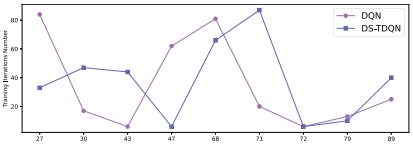


Fig. 7. Training Iters: DS-TDQN vs DQN

settings. Figure 5 shows that a lower ratio contributes to better processing time. From the results, we can find that ZTune (DS-TDQN) has the best performance on most cases, including queries 30, 43, 71, 72, 89. Performance on queries 47, 68, and 79 is also relatively better. Performance on query 27 is worse, but it is still acceptable, which means that the resource is saved much more in this case. In general, the conditional time-resource balance setting contributes to a better processing time compared to other methods due to a heavy penalty on time delay, simultaneously enabling suboptimal resource saving. The experiment results reach the expectation, i.e., time-resource balance with strict delay control.

7.4 Ablation Study

We conduct an ablation study to evaluate DS-TDQN's effectiveness, comparing DS-TDQN with DQN, which uses different methods in searching phase. From Fig. 6 we can find that DS-TDQN achieves better performance on queries 30, 43, 47, 71 and 89, while DS-TDQN and DQN have the same performance on queries 27, 68 and 79. DS-TDQN has worse performance on query 72 with little gap. Generally, DS-TDQN provides better rewards in most cases since dual sampling makes it possible to study more accurate tuning policy for transfer learning, so that a better Spark configuration can be recommended. The improvement of the algorithm is effective. Figure 7 presents the convergence of training DS-TDQN and DQN on different queries. DS-TDQN converges faster on queries 27, 47, 68, and 79, while DQN converges faster on queries 30, 43, 71, and 89. It depends on the probability of state transition and the transfer quality.

8 Related Work

With the development of distributed computing frameworks, a number of methods are proposed to optimize the processing performance. In general, these tuning

methods are divided into six categories, including rule-based approach, cost modeling approach [2,7,12,15], simulated-based approach, experiment-driven approach, machine learning approach [10] and adaptive approach [11,16,18]. In these works, some methods are specially paid attention to and employed in Spark tuning in recent years. Iphicles [7] provides a DCN configuration framework that uses a GNN-based twin performance model for efficient and stable parameter tuning, improving flow completion time with minimal convergence time. [2] proposes a cost model for Spark SQL, providing an error of no less than 14 percents on processing time, while our results achieve a better result on processing time and extend it to resource occupancy index with an acceptable error. [5] proposes a multi-objective optimization model to recommend Spark configuration with a genetic algorithm for configuration updating and Adaboost for performance evaluation. [10] proposes a resource-aware deep learning model with attentional LSTM to predict the performance given the Spark SQL tasks. [6] studies data storage tuning on various hyper-parameters with deep learning and Bayesian optimization. In the most recent, online tuning has become a hotspot in studies. [11] proposes a new tuning framework based on Bayesian Optimization to solve function, cost, and efficiency problems. Some other works pay attention to specific scenarios to conduct the optimization. AutoExecutor [15] concentrates on tuning Spark executor with a machine learning model. Recent works make contribution in segmented fields, while there are still problems for application.

9 Conclusion

In this work, we concentrate on the problem of zero-shot executor tuning on ad hoc queries with balanced requirements of resource and time cost. We introduce ZTune, which integrates SQL plan representation, query performance model, model regression and prediction, and searching with DS-TDQN. ZTune enables high-quality configuration recommendations with SQL plan and history queries. The experiments imply that ZTune achieves better results towards baseline.

Acknowledgment. This work was supported by the National Key R&D Program of China [2024YFF 0617700], the National Natural Science Foundation of China [U23A20309, 62272302, 62502430, 62372296], and the ByteDance Research Project [CT20211123001686, CT20230320002435, CT20241217115379]. Dejun Kong and Xiaofeng Gao are with the Shanghai Key Laboratory of Scalable Computing and Systems. Xiuqi Huang finished most work at Shanghai Jiao Tong University. We also thank Tieying Zhang, Zuzhi Chen, Tiantian Wei, Ron Hu, Rong Kang, Binbin Chen, and Zhaowei Tan for their advice.

References

1. Alipourfard, O., Liu, H.H., Chen, J., Venkataraman, S., Yu, M., Zhang, M.: {CherryPick}: Adaptively unearthing the best cloud configurations for big data analytics. In: USENIX Symposium on Networked Systems Design and Implementation, pp. 469–482. USENIX Association (2017)

2. Baldacci, L., Golfarelli, M.: A cost model for spark sql. IEEE Trans. Knowl. Data Eng. **31**(5), 819–832 (2018)
3. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. J. Mach. Learn. Res. **13**(2), 281–305 (2012)
4. Chen, Y., Goetsch, P., Hoque, M.A., Lu, J., Tarkoma, S.: *d*-simplexed: adaptive delaunay triangulation for performance modeling and prediction on big data analytics. IEEE Transactions on Big Data **8**(2), 458–469 (2019)
5. Cheng, G., Ying, S., Wang, B.: Tuning configuration of apache spark on public clouds by combining multi-objective optimization and performance prediction model. J. Syst. Softw. **180**, 111028 (2021)
6. Dorier, M., et al.: Hpc storage service autotuning using variational-autoencoder-guided asynchronous bayesian optimization. In: 2022 IEEE International Conference on Cluster Computing, pp. 381–393. IEEE (2022)
7. Huang, S., Wang, M., Liu, Y., Liu, Z., Cui, Y.: Iphicles: tuning parameters of data center networks with differentiable performance model. In: 2024 IEEE/ACM 32nd International Symposium on Quality of Service, pp. 1–10. IEEE (2024)
8. Javaid, M.U., Kanoun, A.A., Demesmaeker, F., Ghrab, A., Skhiri, S.: A performance prediction model for spark applications. In: International Conference on Big Data, pp. 13–22. Springer (2020). https://doi.org/10.1007/978-3-030-59612-5_2
9. Kunjir, M., Babu, S.: 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, pp. 1667–1683 (2020)
10. Li, Y., Wang, L., Wang, S., Sun, Y., Peng, Z.: A resource-aware deep cost model for big data query processing. In: IEEE International Conference on Data Engineering, pp. 885–897. IEEE (2022)
11. Li, Y., et al.: Towards general and efficient online tuning for spark. Proc. VLDB Endowment **16**(12), 3570–3583 (2023)
12. Li, Y., Lee, B.C.: Phronesis: efficient performance modeling for high-dimensional configuration tuning. ACM Trans. Archit. Code Optim. **19**(4), 1–26 (2022)
13. Qian, L., Luo, Z., Du, Y., Guo, L.: Cloud Computing: An Overview. In: Jaatun, M.G., Zhao, G., Rong, C. (eds.) CloudCom 2009. LNCS, vol. 5931, pp. 626–631. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10665-1_63
14. Sagaama, H., Slimane, N.B., Marwani, M., Skhiri, S.: Automatic parameter tuning for big data pipelines with deep reinforcement learning. In: IEEE Symposium on Computers and Communications, pp. 1–7 (2021)
15. Sen, R., et al.: Autoexecutor: predictive parallelism for spark SQL queries. Proc. VLDB Endowment **14**(12), 2855–2858 (2021)
16. Shen, Y., et al.: Rover: An online spark sql tuning service via generalized transfer learning. In: ACM Conference on Knowledge Discovery and Data Mining, pp. 4800–4812 (2023)
17. Wu, Y., et al.: Towards resource efficiency: practical insights into large-scale spark workloads at bytedance. Proc. VLDB Endowment **17**(12), 3759–3771 (2024)
18. Xin, J., Hwang, K., Yu, Z.: Locat: Low-overhead online configuration auto-tuning of spark sql applications. In: Proceedings of the 2022 International Conference on Management of Data, pp. 674–684 (2022)