BASE: Bridging the Gap between Cost and Latency for Query Optimization

Xu Chen
University of Electronic Science and
Technology of China
Chengdu, China
xuchen@std.uestc.edu.cn

Zhen Wang
Alibaba Group
Hangzhou, China
jones.wz@alibaba-inc.com

Shuncheng Liu
University of Electronic Science and
Technology of China
Chengdu, China
liushuncheng@std.uestc.edu.cn

Yaliang Li
Alibaba Group
Hangzhou, China
yaliang.li@alibaba-inc.com

Kai Zeng
Alibaba Group
Hangzhou, China
zengkai.zk@alibaba-inc.com

Bolin Ding
Alibaba Group
Hangzhou, China
bolin.ding@alibaba-inc.com

Jingren Zhou
Alibaba Group
Hangzhou, China
jingren.zhou@alibaba-inc.com

Han Su*
University of Electronic Science and
Technology of China
Chengdu, China
hansu@uestc.edu.cn

Kai Zheng*
University of Electronic Science and
Technology of China
Chengdu, China
zhengkai@uestc.edu.cn

ABSTRACT

Query optimization has long been a fundamental yet challenging topic in the database field. Instead of relying on hand-crafted cost models, some recent works have shown the advantages of reinforcement learning (RL) based learned query optimizers. These works often use the cost (i.e., the estimation of cost model) or the latency (i.e., the time of executing the plan) as guidance signals for training their learned models. However, cost-based learning suffers from unsatisfactory latency performance, while latency-based learning is highly time-consuming. In order to bypass such a dilemma, researchers attempt to transfer a learned value network from the cost domain to the latency domain. In this paper, we identify several valuable observations about cost/latency-based training, which motivate us to directly transfer the reward function, instead of the value network. Based on this idea, we propose a two-stage RL-based framework, *BASE*, to bridge the gap between cost and latency. After learning a policy based on cost signals in its first stage, *BASE* formulates the process of transferring the reward function as a variant of inverse reinforcement learning. Intuitively, *BASE* learns to calibrate the reward function and updates the policy regarding the calibrated one in a mutually-improved manner. Extensive experiments exhibit the superiority of *BASE* on two benchmark datasets: Our learned optimizer saves 30% of training time on average to outperform traditional DBMS compared with the time used by SOTA methods. Meanwhile, *BASE* is also further applicable to other learning-based optimizers, and makes these optimizers more efficient, confirming the effectiveness of transferring the reward function.

*Corresponding authors

doi:XX.XX/XXX.XX

1 INTRODUCTION

The query optimizer is a critical component in database management systems (DBMS) and is expected to find the most efficient execution plan for each given SQL query. Recently, studies on machine learning (ML) enhanced DBMS have attracted more and more attention and shown the superiority of boosting database performance in a data-driven way [1, 9, 17]. In particular, reinforcement learning (RL) is applied to produce execution plans and demonstrates its advantages in finding competitive execution plans without heuristics [6, 11, 12, 21].

RL trains a policy in a trial-and-error manner to maximize/minimize the cumulative return of a reward function. When applied to the query optimization, the learned query optimizer generates an execution plan instructed by its current policy and retrieve a reward as feedback indicating how good/bad the plan is. Then the optimizer can update its policy based on the reward to make good plans more likely or bad plans less likely. Thus, the optimizer can learn from a reward function to make its policy better. Generally, two types of signals can be considered in a reward function: *cost* and *latency*. Cost refers to the amount of computation estimation made by the DBMS cost model for an execution plan. Latency, what database users truly care about, is the actual time cost of executing a plan. In fact, recent works have shown the promise of RL-based query optimization. The common part of them is that they learn a value network mapping an execution plan to the cumulative reward (i.e,

the total cost/latency of an execution plan). Thus, they can choose the best plan greedily by searching for the minimum value output from the network. The difference is how they use these two signals for setting reward functions, which can be categorized into three types. The first type of work, such as Rejoin [12], consider only cost signals in reward functions. The second type of work trains their policies with latency signals. For example, Bao [10] trains its value network by latency in a supervised regression manner. Neo [11] pre-trains a value network by regressing the latency signals of demonstrations (i.e., plans generated by traditional query optimizer). The pre-training strategy reduces the number of plans that need to be executed. The third type of work exploits both cost and latency. Both DQ [6] and RTOS [21] pre-train a value network with cost and then transfer it from cost domain to latency domain. They assume the feature extractor (all except for the last layer of the network) can be shared between these two domains so that only the last layer needs to be learned with the latency signals.

Although aforementioned works have shown decent results, we dig deep into them and identify several observations (*OBS*) about training query optimization policies. *OBS-1:* Cost and latency are often inconsistent, and thus the optimal policy regarding cost might not produce the execution plan that results in the least latency. *OBS-2:* Directly learning a policy with latency is time-consuming because it requires a huge amount of time for executing generated plans to collect latency signals, not to mention that some bad plans even take several days to be completed. *OBS-3:* Despite *OBS-1*, only a small portion of queries exhibit a large gap between cost and latency in terms of the behaviors of their corresponding optimal policies. We provide more empirical evidence about these observations in Section 3. *OBS-1* and *OBS-2* raise an effectiveness-v.s.-efficiency dilemma: We need to learn an optimal policy regarding latency rather than cost, but the number of execution has to be minimized for efficient training. According to *OBS-1*, cost-only methods, such as Rejoin, achieve efficient training, but their learned policies are imperfect in terms of latency. The key problem of latency-only methods, such as Neo and Bao, lies in how to reduce the number of plans that need to be executed. The value regression they use needs to be conducted on both good and bad experiences to learn which execution plans should be taken [4]. The pre-trained value network learns from barely good experiences so it still has to explore many plans to improve itself. For DQ and RTOS, they try to achieve efficient training. Even though the feature extractor assumption may hold, learning the randomly-initialized last layer is still inefficient since it has not exploited the experience from the cost domain when performing exploration.

To benefit from both the cost domain and the latency domain, now comes the question: how can we sustain the knowledge learned from the cost domain and improve it to the latency domain? We propose a novel solution that instead of getting rid of the cost signal like previous methods, we can transfer the reward function and policy from the cost domain to the latency domain in a mutually-improved manner. Intuitively, the gap between cost and latency leads to different optimal policies. If we can bridge this gap, namely calibrate "erroneous" cost-based reward function, solving the optimal policy w.r.t. the calibrated reward function will readily result in the policy with the best latency performance. Now comes the second question: what is the benefit of directly working on reward

function instead of value function? According to *OBS-3*, only on a modest portion of queries, cost and latency lead to different optimal plans. By transferring the reward function, we only need to eliminate such conflicts. By contrast, the discrepancy between cost and latency's corresponding optimal value functions is much more significant than the optimal policies, making transferring the value function require more training samples and fine-tuning steps.

Motivated by the benefit of transferring reward function, we propose a two-stage RL-based framework *BASE* to learn a query optimizer. In the first stage of *BASE*, we pre-train a policy with cost signals, which is expected to converge toward the query optimizer of the adopted DBMS efficiently. In the second stage of *BASE*, we transfer the pre-train policy to the latency domain. Inspired by *OBS-3*, *BASE* employs a calibration function for the reward function to fill its gap against latency. The calibration function will correct the cost that leads to bad execution plans in terms of latency. *BASE* formulates the procedure of locating and filling the gap as a variant of inverse reinforcement learning (IRL):

(1) To locate the gap, *BASE* utilizes the active learning (AL) technique to select potential queries and plans on which the current reward function is likely to cause conflicts against latency. As conflicts are discussed in terms of the behavior of corresponding optimal policies, we define two selection criteria—diversity and informativeness, which are measured with the help of the current policy network. Then selected plans will be executed to collect latency signals. This step is analogous to actively soliciting labels for updating a classifier-based reward function in some IRL methods [18].

(2) To fill the gap, the calibration function is updated by encouraging it to correlate the current reward function with latency on newly collected contradictory samples (i.e., query-plan pairs) while preserving their consistency on other samples. Then the current policy will be updated according to the calibrated reward function. This step corresponds to the inner loop of the IRL formulation.

The policy improves the reward function by discovering more conflicts to eliminate. And then the calibrated reward function improves the policy by providing more consistent supervision. By repeating such a procedure, conflicts are gradually discovered and fixed, and the learned calibration function will make the reward function close to latency. Consequently, the learned policy becomes optimal w.r.t. latency.

We conduct extensive experiments to show training efficiency and latency performance of our method. *BASE* shortens 30% of training time on average compared with existing methods to outperform traditional DBMS. With a modest number of query execution, *BASE* reduces latency by 9% more than other methods. Meanwhile, *BASE* is also further applicable to other learning-based optimizers, such as ML-steered query optimizer, and substantially improve training efficiency, which confirms the robustness of *BASE* techniques.

Contributions of this paper are summarized as follows:

- We empirically analyse SOTA RL-based query optimization methods and identify the feasibility and challenges in their cost/latency-based training courses (in Section 3).
- We propose a two-stage RL-based framework *BASE* to efficiently learn an end-to-end query optimizer. The optimizer can predict a complete execution plan for each given query, including join

order, index, and physical operator selection, with satisfactory latency performance (in Section 4).

- To the best of our knowledge, it is the first work to transfer a policy from the cost domain to the latency domain with an IRL formulation. Meanwhile, we make a theoretical analysis to provide the rationale for transferring the reward function (in Section 5).
- We conduct extensive experiments on benchmarks, demonstrating the superiority and practicality of *BASE* (in Section 6).

2 PROBLEM STATEMENT

2.1 Problem Settings

Query Optimization. For a SQL query q, $Rel(q)$ is a set of all base relations in q. Each query execution plan p can be represented by a plan tree. Every leaf node of the tree is a relation $b_i \in Rel(q)$. Every relation b of a specific leaf node is also specified with a scan type $e \in E$, where E represents the set of all scan types, e.g., sequential scan $Seq(b)$ and index scan $Index(b)$. Other non-leaf tree nodes are join implementations $\bowtie_i \in J$, where J represents the set of all join implementations, e.g., nested loop join $\bowtie_N$, merge join $\bowtie_M$, and hash join $\bowtie_H$. p will be executed in a bottom-up order, thus it specifies a join order.

An end-to-end query optimizer QO can map a given SQL query q directly to an execution plan p. The time of executing a plan is often referred to as *latency*. The DBMS cost model can estimate the *cost* for an execution plan, which is expected to reflect the latency of the plan, but they are not positively correlated in practice. In this paper, we study the following objective:

Objective 1. *Given a DBMS, a dataset, and a training workload, our objective is to efficiently learn a QO to replace the cost-based optimizer in DBMS. The learned QO generates the query execution plans of the test workload that have minimal execution latency.*

Domain Transfer. *BASE* improves the training efficiency and stability of a learned query optimizer dramatically by first learning from a data-rich source domain (cost) and then transferring knowledge learned from the source domain to the data-scarce target domain (latency). We defer the formulation of the domain transfer problem in Section 5.1.

2.2 Plan Enumeration in Markov Decision Process

An MDP consists of five components: state space $\mathcal{S} = \{s_t\}$, action space $\mathcal{A} = \{a_t\}$, transition probability distribution $\mathcal{T}(s_{t+1}|s_t, a_t)$, reward $r(s_t, a_t)$ and discount factor γ, which is denoted by $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, r, \gamma)$. For the query optimization problem, we use the same formulation of MDP as previous RL-based works [11, 20].

For an MDP, a policy π is a function that tells an agent (query optimizer) which is the best action to choose in each state. Denoting a *trajectory* as $\tau = \langle s_0, a_0, s_1, \ldots, s_{T-1}, a_{T-1}, s_T \rangle$, the agent is tasked to minimize the expected cumulative reward $\mathbb{E}_{\tau \sim \pi, \mathcal{T}}[R(\tau)] = \mathbb{E}_{a_t \sim \pi(\cdot|s_t), s_{t+1} \sim \mathcal{T}(\cdot|s_t, a_t)}[\sum_{t=0}^{T-1} \gamma^t r(s_t, a_t)]$.

Thus, different reward functions might lead to different optimal policies. For Objective 1, the most straightforward design of the reward function is to let $r(s_t, a_t) = 0, \forall t < T-1$ and $r(s_{T-1}, a_{T-1}) = L(s_T)$, where $L(\cdot)$ stands for a function that maps an input state
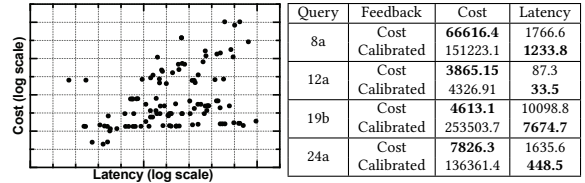

Figure 1: PostgreSQL cost and latency correlation.

Table 1: Discrepancies between cost and latency.

Query	Feedback	Cost	Latency
8a	Cost	**66616.4**	1766.6
	Calibrated	151223.1	**1233.8**
12a	Cost	**3865.15**	87.3
	Calibrated	4326.91	**33.5**
19b	Cost	**4613.1**	10098.8
	Calibrated	253503.7	**7674.7**
24a	Cost	**7826.3**	1635.6
	Calibrated	136361.4	**448.5**

Table 2: Time spent on different training phases in three hours.

Training Signal	Time Consumption (h)			Episodes
	Plan Execution	Neural Network	Get Cost	
Latency	2.76	0.24	0	226
Cost	0	2.89	0.11	88,524

s_t to the latency signal of its corresponding plan (i.e., the plan l_t represents). This setting is widely used in previous methods [11, 21]. Obviously, with such a reward function, the cumulative reward can be minimized by generating the plan of least latency. However, it is time-consuming for model training. The reward function needs to be designed carefully to achieve the Objective 1. We elaborate more about reward function design in Section 4.

3 OBSERVATIONS IN QUERY OPTIMIZATION

OBS-1: Cost is inconsistent with latency, which leads to different optimal policy. As an example, we execute queries from the Join Order Benchmark (JOB) [7] workload using a traditional DBMS, i.e., PostgreSQL [19]. Figure 1 shows the correlation between cost estimation and execution latency in PostgreSQL. Obviously, cost signals do not align with latency signals. Cost is designed to reflect the relative latency performance, while in reality, it could produce execution plans with long-running time.

In another example, we train the value model proposed in [11] based on two signals: the cost from PostgreSQL and the calibrated cost from *BASE*. Then we use these two trained models to generate execution plans for the JOB workload and compare their corresponding performance. As shown in Table 1, the cost-based model generates execution plans with lower cost, which indicates it learns a policy w.r.t. cost signals. On the contrary, our fine-tuned model generates plans with higher cost but with lower latency performance. This contradiction shows the inconsistency between cost and latency and its influence on RL policy learning.

OBS-2: The time consumption of purely latency-based training in RL is catastrophic. As an example, we compare the efficiency of training a value model on JOB based on cost and latency, where the query execution timeout is set to 90s. Both latency-based and cost-based model is trained for 3 hours, respectively. As shown in Table 2, 92% of time is spent on query plan execution for latency-based training, and the model is trained for only 226 episodes. For comparison, 96% of time is used for training the neural network for cost-based training, and it is trained for 88,524 episodes. In the end, the cost-based model shows a much more robust performance than a latency-based model in a limited training time since cost-based training covers much more candidate query execution plans.

OBS-3: Although OBS-1 holds, only a small portion of queries exhibit a large gap between cost and latency in terms of the behaviors of their corresponding optimal policies. We train two value models with cost and latency serving as the respective feedback for a sufficient amount of time until convergence to investigate the impact of the gap. Then, we allow two value models to serve as greedy cost- and latency-based policies to produce query execution plans. Accordingly, less than 5% of execution plans generated by cost-based policy leads to long-running plans compared to plans generated by latency-based policy. This result proves that the cost-based policy is close to the optimal policy in terms of latency. Thus, compared to the enormous state-action space, only a limited number of cost signals should be calibrated to produce the same optimal policy as the latency does.

4 FRAMEWORK OVERVIEW

Based on *OBS*, we propose a two-stage RL-based framework, namely *BASE*. Conceptually, *BASE* conducts transfer for RL from the source (cost) domain to the target (latency) domain. In the jargon of RL, the tasks to be solved are two MDPs, namely $\mathcal{M}^{(c)} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, r^{(c)}, \gamma)$ and $\mathcal{M}^{(l)} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, r^{(l)}, \gamma)$, where $r^{(c)}$ and $r^{(l)}$ denote cost-based and latency-based reward functions, respectively. The workflow of *BASE* is shown in Figure 2.

Stage 1: In first stage, *BASE* solves $\mathcal{M}^{(c)}$ for its optimal policy $\pi^{(c)*}$. we adopt reward shaping [14] and define our cost-based reward function as follows:

$$r^{(c)}(s_t, a_t) = C(s_{t+1}) - C(s_t), \tag{1}$$

for $t = 1, \ldots, T-1$. It is also worth noticing that $C(s_0) = 0$, since the initial state corresponds to an empty plan. To solve $\mathcal{M}^{(c)}$, we can learn its optimal policy, its optimal value function, or both of them, using corresponding RL algorithms as existing works [6, 11, 21]. For ease of transfer, we choose a policy-based algorithm to learn the optimal policy of $\mathcal{M}^{(c)}$ since it avoids learning the value function $Q^\pi(s, a)$ explicitly. Specifically, we employ a deep policy network to represent a stochastic policy from which we can sample actions for each input state: $a \sim \pi_\theta(\cdot|s)$. We adopt the same neural architecture as Neo [11] to encode each input state.

Stage 2: In the second stage, *BASE* starts with $r^{(c)}$ and $\pi^{(c)*}$, and transfers towards $r^{(l)}$ and the corresponding optimal policy $\pi^{(l)*}$, in a mutually-improved manner. Eventually, the learned policy can be applied as an end-to-end query optimizer that optimizes latency performances. We describe details of stage 2 in Section 5.

5 REWARD FUNCTION CALIBRATION

5.1 Formulation

As defined in Section 4, the only difference between $\mathcal{M}^{(c)}$ and $\mathcal{M}^{(l)}$ lies in their reward functions. In the first stage of *BASE*, $r^{(c)}$ is defined in Eq. (1). $r^{(l)}$ can be trivially defined according to Section 2 to encourage a policy that minimizes the latency of its generated plan. Then we study the condition under which $r^{(c)}$ and $r^{(l)}$ lead to the same optimal policy.

LEMMA 1. *Denoting the set of the terminal states (i.e., the states corresponding to complete execution plans) that we can traverse from s_0 by $U(s_0)$, $r^{(c)}$ and $r^{(l)}$ lead to the same optimal policy, if $\forall s_0 \in \mathcal{S}, \arg\min_{s \in U(s_0)} C(s) = \arg\min_{s \in U(s_0)} L(s)$.*
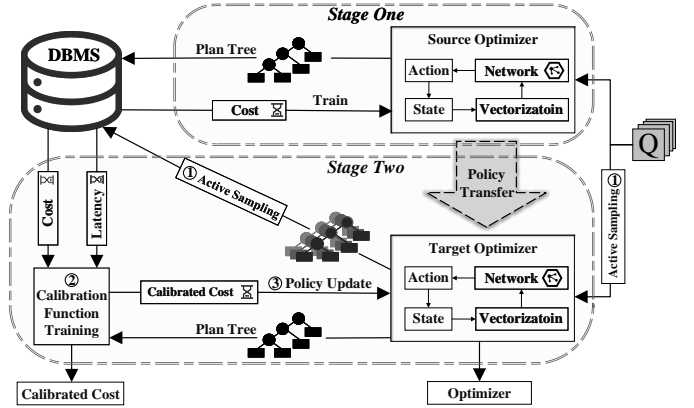

Figure 2: Framework overview.

In reality, this condition cannot be satisfied perfectly (see *OBS-3*), where $\exists s_0, S_T = \arg\min_{s \in U(s_0)} L(s)$, but $\exists S'_T \in U(s_0)$, s.t., $C(S'_T) < C(S_T)$. We aim to eliminate such discrepancies by calibrating $r^{(c)}$ so that the relationship between the cumulative rewards of the corresponding trajectories of S_T and S'_T becomes consistent with $r^{(l)}$. To this end, we propose to represent a calibrated reward function by applying a parameterized calibration function $g_\phi(\cdot)$ to the cost signals $C(\cdot)$ and following Eq. (1):

$$r_\phi(s_t, a_t) = g_\phi(s_{t+1})C(s_{t+1}) - g_\phi(s_t)C(s_t), \tag{2}$$

where $g_\phi : \mathcal{S} \to \mathbb{R}$ plays the role of cost model calibration and is to be learned for making r_ϕ consistent with $r^{(l)}$.

Note that our calibration function only makes the calibrated cost signals and the latency signals more *correlated* (satisfying Lemma 1), which is easier than making the prediction accurate. This goal is fundamentally different from previous RL-based learned query optimizers [11, 20, 21]. They all regard query optimization as a regression task i.e., minimizing the mean square error between prediction from value functions and actual latency. Latency prediction will cause dramatic changes to the pre-trained model, as cost and latency have different numeric ranges and basic statistics. As a result, it causes large fluctuations in query optimization performance. This makes transferring the value function and policy require more training samples and fine-tuning steps. We show the experimental results in Sections 6.3 and 6.4

BASE formulates the procedure of locating and filling the gap between r_ϕ and $r^{(l)}$ as a variant of IRL. As shown in Figure 3, *BASE* maintains π_θ and r_ϕ which, at the beginning of transfer, are close to $\pi^{(c)*}$ and $r^{(c)}$, respectively. Then *BASE* utilizes π_θ to actively locate discrepancies for updating r_ϕ. And π_θ is updated according to the latest r_ϕ. These steps are alternatively repeated until the convergence of π_θ and r_ϕ. With the discrepancies between r_ϕ and $r^{(l)}$ being gradually eliminated, π_θ is ensured to be close to $\pi^{(l)*}$.

5.2 Algorithm

As a transfer for RL, we collect *experience* by using the maintained policy π_θ to interact with $\mathcal{M}^{(c)}$ and $\mathcal{M}^{(l)}$. We first introduce how to organize the collected experience. For each sampled trajectory
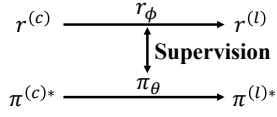
Figure 3: Transfer procedure.

$\tau : s_0, a_0, \ldots, a_{T-1}, s_T$, we store every $(s_t, C(s_t), L(s_t))$ triplet in buffer $\mathcal{B}_C$. We also calculate the cumulative rewards $R^{(c)}(\tau)$ and $R^{(l)}(\tau)$ based on such triplets and store the $(\tau, R^{(c)}(\tau), R^{(l)}(\tau))$ triplet in buffer $\mathcal{B}_T$. For any two collected trajectories τ and $\tau' \in \mathcal{B}_T$, if $(R^{(l)}(\tau) - R^{(l)}(\tau'))(R^{(c)}(\tau) - R^{(c)}(\tau')) < 0$, i.e., the condition in Lemma 1 is violated, we call the trajectory pair (τ, τ') a *discrepancy*. Otherwise, we call (τ, τ') a *preservation*. Then we define $C_d = \{(\tau, \tau', R^{(c)}(\tau), R^{(c)}(\tau'))|\tau, \tau' \in \mathcal{B}_T, (\tau, \tau')$ is a discrepancy $\}$ and its counterpart for preservation C_p. For the ease of discussion, we assume $R^{(l)}(\tau) < R^{(l)}(\tau')$ for each tuple in C_d or C_p, which avoids the consideration of both (τ, τ') and (τ', τ).

In the beginning, all these buffers are initialized as an empty set $\emptyset$. The pre-trained policy π_θ is adopted as an initialization and to be updated, and the calibration function g_ϕ is initialized to output around 1 for any input state so that $r_\phi(s, a) = r^{(c)}(s, a)$. After that, *BASE* trains the learned query optimizer iteratively, as shown in Figure 2. In every iteration, *BASE* contains three steps: ① Active Sampling: we first use our active sampling module to sample queries from the training workload and corresponding execution plans, where the sample criteria are based on the current policy π_θ. Then we collect the aforementioned experiences by executing the selected queries and their corresponding execution plans. ② Calibration Function Training: Then we update the reward function r_ϕ based on experiences collected in $\mathcal{B}_C$, C_d, and C_p. ϕ is updated by minimizing an objective function $\mathcal{L}$ to encourage r_ϕ to eliminate the discrepancy while maintaining the preservation. ③ Policy update: Lastly, we fine-tune our policy π_θ according to the latest reward function r_ϕ. Intuitively speaking, in each iteration, we push the maintained reward function r_ϕ and policy π_θ towards $r^{(l)}$ and $\pi^{(l)*}$. We repeat such a procedure for at most M iterations, where any convergence or early-stop criterion can be easily incorporated.

5.3 Active Sampling

In the sampling phase, we first sample queries from the training workload Q and then sample valuable execution plans of the sampled queries. We define two criteria for query sampling and plan sampling: *informativeness* and *diversity*. Informativeness helps the current policy locate the discrepancy efficiently. Denoting the trajectory of an execution plan by τ, we define the following informativeness score:

$$S_{\pi_\theta}(\tau) = -\sum_{t=1}^{T-1} \sum_{a \in \mathcal{A}(s_t)} \pi_\theta(a|s_t) \log(\pi_\theta(a|s_t)), \quad (3)$$

where the policy entropy $-\sum_{a \in \mathcal{A}(s)} \pi_\theta(a|s) \log(\pi_\theta(a|s))$ indicates how certain it is to choose an action at the state s. Diversity of sampled queries and plans helps stabilize the learning dynamics of our calibrated reward function, which will be explained later.

Query Sampling. We employ the weighted K-means algorithm [23] to sample queries. Specifically, we first extract the embedding representation of each query $q \in Q$ from the intermediate layer of π_θ that embeds q into a latent space. Then each query q is represented by its embedding for the clustering algorithm. To ensure the informativeness, each query is weighted based on its execution plan τ greedily generated by π_θ. We use informativeness score $S_{\pi_\theta}(\tau)$ to serve as the weight of the query. Since τ is decided by π_θ with the largest probability, $S_{\pi_\theta}(\tau)$ reveals the uncertainty of π_θ for the current query q. To ensure the diversity, after $|Q|$ queries being clustered into $\lfloor k\% \times |Q| \rfloor$ clusters, each query closest to the centroid of a specific cluster will be selected. Finally, the $\lfloor k\% \times |Q| \rfloor$ selected queries form the mini-batch used in current iteration.

Plan Sampling. To ensure diversity, we use Monte Carlo dropout [13] by plugging dropout layers into π_θ. Our pre-trained policy tends to be deterministic, which results in a narrow range of samples and thus cannot effectively explore the target domain without the dropout mechanism. Specifically, for each sampled query, we sample its plans from π_θ multiple times with a drop-out layer. The dropout mechanism will turn off some neurons randomly at each time, thus it gives diverse results. To ensure informativeness, then we select the plan with the largest informativeness score.

After that, we will execute the selected query execution plans and get cost and latency feedback for each corresponding trajectory $\tau : s_0, a_0, \ldots, a_{T-1}, s_T$. As a result, we collect $(s_t, C(s_t), L(s_t))$ triplet in $\mathcal{B}_C$. $\mathcal{B}_T$ stores the trajectory and corresponding accumulated reward $(\tau, R^{(c)}(\tau), R^{(l)}(\tau))$. For every newly selected trajectory τ, we will compare it to existing ones (i.e., every $\tau' \in \mathcal{B}_T$). If the following inequality is satisfied:

$$(R^{(l)}(\tau) - R^{(l)}(\tau'))(R^{(c)}(\tau) - R^{(c)}(\tau')) < 0,$$

then (τ, τ') is a discrepancy, and we add it to C_d. Discrepancies from different queries allow r_ϕ to learn from the shared partial plans, which is helpful for its generalization. If there is no discrepancy, we also want to preserve the consistency, so we add that pair to C_p.

5.4 Calibration Function Training

After collecting discrepancies and preservation, we aim to update ϕ so that r_ϕ leads to the same optimal policy as $r^{(l)}$. We propose to gradually tune our reward function by eliminating the discrepancies while keeping the preservation.

For the discrepancies, we aim to update ϕ so that $\forall((\tau, \tau', R^{(c)}(\tau), R^{(c)}(\tau')) \in C_d, R_\phi(\tau) \leq R_\phi(\tau')$. However, as a parameterized function $g_\phi(\cdot)$, the updates may simultaneously bring in "wrong" reward values on other state-action pairs. Therefore, $g_\phi(\cdot)$ is also expected to keep the consistency with $r^{(l)}$ for the preservation. To this end, we define a hinge loss as follow:

$$\mathcal{L}^{(1)}(\phi) = \sum_{(\tau, \tau') \in C_d \cup C_p} \max(0, R_\phi(\tau) - R_\phi(\tau') + \delta), \quad (4)$$

where $\delta > 0$ is a pre-specified margin. In our implementation, we use all elements from C_d and sample the recently added elements of C_p with higher priority.

Furthermore, locating the discrepancy becomes even more inefficient, along with the improvement of our policy. To further improve the sample efficiency in calibration function training, we present the following proposition to motivate another objective function.

PROPOSITION 1. *If the calibrated cost signals $g_\phi(\cdot)C(\cdot)$ and the latency signals $L(\cdot)$ are linearly positive correlated, r_ϕ leads to the same optimal policy as $r^{(l)}$.*

Thus, we define our another objective function as follow:

$$\mathcal{L}^{(2)}(\phi) = \sum_{s \in \mathcal{B}_c} |\text{CorrCoef}[g_\phi(s)C(s), L(s)] - 1|, \quad (5)$$

where $\text{CorrCoef}(\cdot, \cdot)$ is Pearson product-moment correlation coefficient, which is adopted to measure the linear relationship between $g_\phi(s)C(s)$ and $L(s)$. We want it to be as close to one as possible.

We combine the constraint objective (Eq. (4)) and the linear relationship objective (Eq. (5)) as the final objective $\mathcal{L}$:

$$\mathcal{L}(\phi) = \lambda \mathcal{L}^{(1)}(\phi) + \mathcal{L}^{(2)}(\phi), \quad (6)$$

where $\lambda \geq 0$ is a factor to control the penalty of samples that violate the constraints. We gradually increase λ during the training iterations. In training our calibration function, we update ϕ by minimizing this $\mathcal{L}(\phi)$.

6 EXPERIMENTS

6.1 Experiment Setup

We evaluate our model on two widely-used benchmarks. The datasets are split into training and testing parts based on previous work [11]: **Join Order Benchmark (JOB) [7]:** a real-world dataset based on IMDB to provide realistic workloads. It has 113 queries from 33 templates. It has 3.6GB of data (11GB when counting indexes) and 21 tables. The number of relations in each query ranges from 4 to 17. **STACK [10]:** STACK consumes 100GB storage space. We adopt 500 queries from 16 templates. The number of relations in each query ranges from 4 to 12.

Compared Methods: *BASE* is trained in two stages, namely policy pre-training (denoted by **Pp**) and reward function calibration (denoted by **Rfc**). We compare *BASE* with the following methods:

- **PG:** The vanilla cost-based optimizer of PostgreSQL [19]. The optimizer is based on dynamic programming.
- **Neo:** The end-to-end learned optimizer proposed in [11]. The Neo is trained in two stages: (1) It is first trained by execution plans provided by PG (denoted by **Lfd**). (2) It is further trained by the RL algorithm (denoted by **Lt**).
- **Balsa:** The end-to-end learned optimizer proposed in [20]. For a fair comparison, we bootstrap Balsa from PG. And we use a local model instead of a cluster model as the same hardware resource as other baselines.
- **LO:** The Neo trained by latency without any pre-training. It provides a baseline for all learned optimizers.

To study the efficiency of different transfer methods that transfer a pre-trained policy based on the cost to a better policy based on latency, we fix the pre-trained value/policy model in the first stage and vary the method for the second stage. We denote the second stage of our method as *BASE*-Rfc. The variants are listed as follows:

- **Direct Transfer:** We change the feedback directly from cost to latency for the second stage. In the experiment, we consider two variants of direct transfer: value-based (**Q-Late**) algorithm and policy-based (**P-Late**) algorithm.
- **RTOS (Q-RTOS):** It uses the multi-task learning method proposed in [21] for the second stage.

- **DQ (Q-DQ):** It uses the inductive transfer learning method proposed in [6] for the second stage.
- **Classifier-Based Reward Function (P-Clf):** It uses a classifier-based reward function proposed in [3] for the second stage.

Evaluation Metrics:

- **Geometric Mean Relevant Latency (GMRL) [21]:** $GMRL = \prod_{i=1}^{n} \frac{Latency(q)}{Latency_{PG}(q)}$. The lower the numerical value is, the better the latency performance is compared to PG.
- **Jumpstart Performance (JP) and Asymptotic Performance (AP):** The initial performance and the ultimate performance of the learned query optimizer at the latency training stage.
- **Performance with Fixed Demonstration Amount (PA):** Final performance within certain training examples.
- **Transfer Ratio (TR):** The amount of performance increases in a certain period of time i.e., increasing from JP to PA. The larger the TR is, the better the transferability is.

6.2 Overall Performance

Table 3: GMRL comparison within 10 hours training time.

GMRL Methods / Workloads	JOB	STACK
LO	2.22	1.24
Neo	0.85	0.94
Balsa	0.69	0.75
BASE	**0.64**	**0.61**

To show the overall performance, we conduct end-to-end experiments on two benchmarks from the aspects of latency performance and training efficiency. Specifically, all algorithms are trained within 10 hours for a fair comparison. During that time, *BASE* and Balsa model is trained on cost for 50000 iterations (about two hours). Given the execution plans provided by PG, Neo is trained on demonstrations for 1000 iterations (around two hours). **Latency Performance.** To measure the latency performance, Table 3 shows GMRL of all methods. Overall, *BASE* outperforms all the other methods including PG on these two benchmarks: For the JOB workload, *BASE* improve Balsa by 7% Neo by 24%, PG by 36% and LO by 71% within the same training time. For the STACK workload, *BASE* outperforms Balsa by 18%, Neo by 35%, PG by 4% and LO by 50%. It is worth noting that STACK is more complex and has larger storage space. It takes a long time for model training.

Training Efficiency. Table 4 exhibits the time that a learned optimizer needs to take to outperform the traditional optimizer, PG, which is when GMRL=1 in this experiment. We observe that:

(1) Among all algorithms, *BASE* is the fastest one. *BASE* takes 94% and 81% training time of Balsa on JOB and on STACK to make the optimizer reach the threshold. *BASE* only takes 57% and 56% training time of Neo on JOB and on STACK to make the optimizer reach the threshold. In our experiment, LO cannot achieve PG performance in 10 hours, which reflects its low training efficiency.

(2) In the second stage, *BASE* shows better efficiency in running more episodes per hour (e/h in Table 4). This is beneficial from the active sampling module. Instead of sampling greedily w.r.t the current policy, *BASE* samples more diverse query execution plans,

Table 4: Time used to reach GRML=1.

Workloads	Methods	Pre-training		Fine-tuning			Total Time
		episode	time (h)	episode	time(h)	e/h	
JOB	LO	0	0	1650	10	165	10+
	Neo	1000	2.09	1180	6.13	192	8.22
	Balsa	50000	2.05	614	2.97	207	5.02
	BASE	50000	2.02	590	2.71	**218**	**4.73**
STACK	LO	0	0	1890	10	189	10+
	Neo	800	1.58	1050	6.02	174	7.60
	Balsa	50000	2.01	602	3.22	187	5.23
	BASE	50000	2.03	470	2.24	**210**	**4.27**

Table 5: Transfer methods comparison.

| Workloads	Methods	JP	PA	TR	AP
JOB	Q-Late	1.20	1.87	-55.83%	0.60
	Q-RTOS	1.15	0.82	28.70%	0.56
	Q-DQ	1.14	1.87	-64.04%	0.94
	P-Clf	0.99	0.97	2.02%	0.91
	P-Late	0.97	0.84	13.40%	0.55
	BASE-Rfc	**0.97**	**0.66**	**31.96%**	**0.53**
STACK	Q-Late	1.16	0.96	17.24%	0.71
	Q-RTOS	1.18	0.75	36.44%	0.56
	Q-DQ	1.13	1.08	4.42%	0.88
	P-Clf	1.09	0.67	38.34%	0.74
	P-Late	1.05	0.75	28.57%	0.62
	BASE-Rfc	**1.03**	**0.52**	**49.51%**	**0.43**

which makes the query optimizer unlikely to be trapped in suboptimal plans for a long time. This helps the query optimizer explore potential execution plans efficiently.

6.3 Evaluation of Transfer Strategies

With the help of cost training, the learned optimizer can imitate the traditional DBMS behaviors after the first stage. To examine how far the learned optimizer can go beyond the traditional DBMS in the second stage, we compare five transfer methods in previous works (in Section 6.1) with *BASE*-Rfc. For a fair comparison, in the first stage, we train two types of learned optimizers (i.e., value-based and policy-based) until convergence to achieve similar performance as PG in terms of cost. And then we transfer the first-stage model to adapt to the latency environment. All methods are implemented with the same number of query executions. We show the evaluation metrics in Table 5.

(1) The transferability can be ranked as *BASE*-Rfc > Q-RTOS > P-Late ≈ P-Clf > Q-Late ≈ Q-DQ according to TR in Table 5. This shows the superiority of *BASE*-Rfc in terms of effectiveness and efficiency. *BASE*-Rfc and Q-RTOS have better performance because they can absorb more knowledge from the source domain (i.e., cost). So the near-oracle policy in the source domain helps them transfer easier in the target domain (i.e., latency). It can be seen that Q-Late and Q-DQ almost fail in the fine-tuning phase. This is expected because taking latency as feedback can result in drastically different optimal Q values compared to taking cost as feedback, which causes hindrance to transfer learning.

(2) As for JP, both policy-based RL and value-based RL are feasible for cost training. They achieve similar performance before transferring. However, policy-based RL is more suitable for transfer learning: Their TR (19.08%) is better on average compared to the TR of value-based methods (−10.63%).
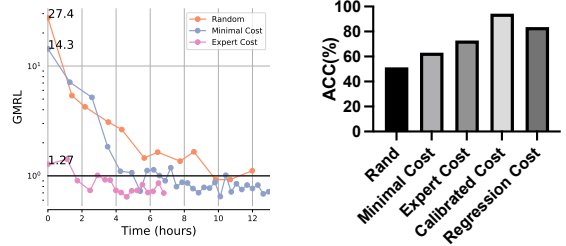
(a) Learned query optimizers pre-trained by different cost model.

(b) Accuracy of different cost models.

Figure 4: Effectiveness of calibration function training.

(3) As for AP, as long as the training time is long enough, all learned query optimizers can achieve better performance than PG (All GMRL results are less than 1), which shows the upper limit of RL-enhanced DBMS.

6.4 Evaluation of Design Choices

Calibration Function Training. To test the transferability of *BASE*, we conduct an empirical study of transferring different pre-trained query optimizers on the JOB workload. We will use three pre-trained query optimizers: (i) a query optimizer initialized randomly (**Rand**) (ii) a query optimizer pre-trained by a minimal cost model (**Minimal Cost**: cardinality estimator from PG) (iii) a query optimizer pre-trained by an expert cost model (**Expert Cost**: default cost model from PG). After the pre-training stage, we conduct the second stage in *BASE*: We calibrate the cost model and progressively train the learned query optimizer with the cost model.

Figure 4(a) shows the training curve of the different learned query optimizers. It is obvious that the randomly initialized query optimizer has a bad performance at the beginning (27×). And it takes about 10 hours to outperform PG. The learned optimizer pre-trained with a minimal cost model has a much better performance at the beginning (14×). It takes much less time to exceed PG. The learned optimizer pre-trained with an expert model achieves almost the same performance as PG (1.27×) and it takes less than 2 hours to outperform PG. In conclusion, *BASE* can fine-tune the different pre-trained or even random initialized learned query optimizers tailored to certain workloads or hardware. However, different pre-training methods do make a big difference in fine-tuning efficiency.

Based on the collected experience buffer, we calibrate the expert cost model with two different methods: **Calibrated Cost** (our proposed hinge loss) and **Regression Cost** (commonly used regression loss: mean square error). Figure 4(b) shows the accuracy of the different cost models. The accuracy indicates whether the calibrated cost model has a discrepancy with latency for every trajectory pair (τ, τ'). It can be shown that the accuracy of those cost models ranks as Calibrated Cost (94%) > Regression Cost (83%) > Expert Cost (72%) > Minimal Cost (61%) > Random Cost (51%). It empirically proves that our proposed correlation learning goal is much more effective than the regression goal.

Active Sampling. To show the effectiveness of plan sampling in our active sampling (AS) module, we conduct a micro-benchmark
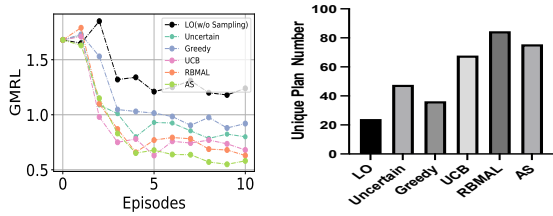
(a) Use different active learning (b) Unique plan number for different method to replace active sampling. active learning methods.

Figure 5: Effectiveness of plan sampling.

Figure 6: Applying *BASE* to ML-steered query optimizer.

about the active learning methods. Specifically, we reuse the optimizer trained on JOB and then train it on the new workload called Ext-JOB [11]. We use different active learning methods to pick valuable training queries and physical plans to collect in our experience buffer. All methods are trained for 10 episodes. We compare with the following methods: **LO**: same as the previous definition. Uncertain [8]: Select potential training query and plans with the highest uncertainty score. **Greedy**: Select potential training queries randomly and then generate physical plans greedily based on the current query optimization policy. **UCB** [5]: A common exploration strategy used in RL problems. **RBMAL** [2]: A SOTA active learning method that combines the uncertain score with a weighted distance of each sample data to the labeled training data.

Figure 5(a) shows the training curve of different methods used to train the learned query optimizer. The effectiveness of different active learning methods can be ranked as AS > RBMAL > UCB > Uncertain > Greedy > LO. AS perform best as it ensures both diversity and informativeness of the potential queries and plans. RBMAL performance is pretty close to AS. However, the overhead of calculating data point distance is quadratic to the number of data points. Thus, in practice, *BASE* adopt AS as a more efficient method.

Figure 5(b) shows the average number of unique plans in one episode during the policy update phase. Especially for a mature learned query optimizer, the policy tends to be deterministic, which results in a narrow range of samples. For every episode, the AS module samples a diversified batch of query execution plans as valuable experience. The diversified experience helps the query optimizer to explore diversified physical plans during policy updates. Thus, the unique plans generated from AS are much more than that of LO, greedy, and uncertain methods.

6.5 Applying *BASE* to ML-steered Optimizer

DBMSes provide hint sets (e.g, disable loop join) to fine-tune the behavior of the corresponding query optimizer. Bao [10], a recently proposed learned optimizer that learns to choose hint sets. Similar to previous work, it trains a predictive model to estimate plan performance. *BASE* domain transfer technique can also be applied to training the predictive from a data-rich knowledge base instead of an empty knowledge base. *BASE* substantially optimizes Bao source code [16] in two aspects (denoted by *BASE* +Bao): Firstly, *BASE* initialize the Bao predictive model following Eq (2) from
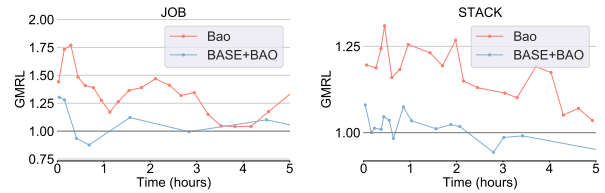
cost model. Secondly, *BASE* train a predictive model based on our calibration function training strategy to transfer the knowledge.

Figure 6 shows the training procedure of our optimization on two datasets. It is not surprising to see that *BASE* +Bao has significantly better initial performance (GMRL On JOB: *BASE* +Bao: 1.30, Bao: 1.44; GMRL On STACK: *BASE* +Bao: 1.08, Bao: 1.19) and then reach the peak performance much faster (Time on JOB:*BASE* +Bao: 2.9h, Bao: 3.8h; Time on STACK: *BASE* +Bao: 2.1h, Bao: 5+h). It demonstrates that *BASE* can enhance learning efficiency to achieve better performance for the learning-based query optimizer.

7 RELATED WORK

Neo [11] builds an end-to-end query optimizer that produces complete execution plans. However, Neo is trained completely based on latency signals, which requires DBMS to execute numerous plans including potentially bad ones. Some other similar works include Rejoin [12], DQ [6], and RTOS [21] leverage cost as a trade-off to increase training efficiency and then transfer the pre-trained model based on the cost to a new model that can adapt to latency signals. Since the feature representations are the same in source and target domain for the query optimization settings, DQ and RTOS leverage inductive transfer learning methods [15] that change representations in the output layer. Bao [10] and QO-Advisor [22] steer the traditional query optimizer by tuning hint sets. They learn a predictive model to estimate the latency of a plan generated by the query optimizer. Instead of training the ML model from an empty or randomly filled knowledge base, *BASE* calibrate the ML model by transferring domain knowledge from cost to latency.

8 CONCLUSIONS

In this paper, we propose *BASE*, a two-stage RL-based framework that bridges the gap between cost and latency for learning an end-to-end query optimizer with satisfactory latency performances. We conduct extensive experiments on two public benchmarks to offer evidence that *BASE* has better training efficiency and transferability than SOTA methods. Meanwhile, *BASE* is also applicable to ML-steered optimizers for better training efficiency.

REFERENCES

[1] Debabrota Basu, Qian Lin, Weidong Chen, Hoang Tam Vo, Zihong Yuan, Pierre Senellart, and Stéphane Bressan. 2015. Cost-model oblivious database tuning with reinforcement learning. In *Database and Expert Systems Applications*. 253–268.

[2] Thiago NC Cardoso, Rodrigo M Silva, Sérgio Canuto, Mirella M Moro, and Marcos A Gonçalves. 2017. Ranked batch-mode active learning. *Information Sciences* 379 (2017), 313–337.

[3] Jonathan Ho and Stefano Ermon. 2016. Generative adversarial imitation learning. *Advances in neural information processing systems* 29 (2016).

[4] Julian Ibarz, Jie Tan, Chelsea Finn, Mrinal Kalakrishnan, Peter Pastor, and Sergey Levine. 2021. How to train your robot with deep reinforcement learning: lessons we have learned. *The International Journal of Robotics Research* 40, 4-5 (2021), 698–721.

[5] Emilie Kaufmann, Olivier Cappé, and Aurélien Garivier. 2012. On Bayesian upper confidence bounds for bandit problems. In *Artificial intelligence and statistics*. PMLR, 592–600.

[6] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196* (2018).

[7] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[8] David D Lewis and Jason Catlett. 1994. Heterogeneous uncertainty sampling for supervised learning. In *Machine learning proceedings 1994*. Elsevier, 148–156.

[9] Lin Ma, Bailu Ding, Sudipto Das, and Adith Swaminathan. 2020. Active learning for ML enhanced database systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 175–191.

[10] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2022. Bao: Making learned query optimization practical. *ACM SIGMOD Record* 51, 1 (2022), 6–13.

[11] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. *Proceedings of the VLDB Endowment* 12, 11 (2019).

[12] Ryan Marcus and Olga Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–4.

[13] Kristian Miok, Dong Nguyen-Doan, Daniela Zaharie, and Marko Robnik-Šikonja. 2019. Generating data using Monte Carlo dropout. In *2019 IEEE 15th International Conference on Intelligent Computer Communication and Processing (ICCP)*. IEEE, 509–515.

[14] Andrew Y Ng, Daishi Harada, and Stuart Russell. 1999. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, Vol. 99. Citeseer, 278–287.

[15] Sinno Jialin Pan and Qiang Yang. 2009. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering* 22, 10 (2009), 1345–1359.

[16] RyanMarcus. 2021. BAO source code. https://github.com/learnedsystems/BaoForPostgreSQL.

[17] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. The case for automatic database administration using deep reinforcement learning. *arXiv preprint arXiv:1801.05643* (2018).

[18] Avi Singh, Larry Yang, Kristian Hartikainen, Chelsea Finn, and Sergey Levine. 2019. End-to-End Robotic Reinforcement Learning without Reward Engineering. *environment (eg, by placing additional sensors)* 34 (2019), 44.

[19] The PostgreSQL Global Development Group. 2022. PostgreSQL 10 Documentation. https://www.postgresql.org/docs/10/index.html.

[20] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *Proceedings of the 2022 International Conference on Management of Data*. 931–944.

[21] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement learning with tree-lstm for join order selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1297–1308.

[22] Wangda Zhang, Matteo Interlandi, Paul Mineiro, Shi Qiao, Nasim Ghazanfari, Karlen Lie, Marc Friedman, Rafah Hosn, Hiren Patel, and Alekh Jindal. 2022. Deploying a steered query optimizer in production at Microsoft. In *Proceedings of the 2022 International Conference on Management of Data*. 2299–2311.

[23] Fedor Zhdanov. 2019. Diverse mini-batch active learning. *arXiv preprint arXiv:1901.05954* (2019).