A Sampling-based Learning Framework for Big Databases

Jingtian Zhang
11421015@zju.edu.cn
Zhejiang University
Hangzhou, Zhejiang, China

Sai Wu
wusai@zju.edu.cn
Zhejiang University
Hangzhou, Zhejiang, China

Junbo Zhao*
j.zhao@zju.edu.cn
Zhejiang University
Hangzhou, Zhejiang, China

Zhongle Xie
xiezl@zju.edu.cn
Zhejiang University
Hangzhou, Zhejiang, China

Feifei Li
lifeifei@alibaba-inc.com
Alibaba Cloud
Hangzhou, Zhejiang, China

Yusong Gao
jianchuan.gys@alibaba-inc.com
Alibaba Cloud
Hangzhou, Zhejiang, China

Gang Chen
cg@zju.edu.cn
Zhejiang University
Hangzhou, Zhejiang, China

ABSTRACT

The autonomous database of the next generation aims to apply the reinforcement learning (RL) on tasks like query optimization and performance tuning with little or no human DBAs' intervention. Despite the promise, to obtain a decent policy model in the domain of database optimization is still challenging — primarily due to the inherent computational overhead involved in the data hungry RL frameworks — in particular on large databases. In the line of mitigating this adverse effect, we propose *Mirror* in this work. The core to *Mirror* is a sampling process built in an RL framework together with a transferring process of the policy model from the sampled database to its original counterpart. While being conceptually simple, we identify that the policy transfer between databases involves heavy noise and prediction drifting that cannot be neglectable. Thereby we build a theoretical-guided sampling algorithm in *Mirror* assisted by a continuous fine-tuning module. The experiments on the PostgreSQL and an industry database PolarDB validate that *Mirror* has effectively reduced the computational cost while maintaining a satisfactory performance.

CCS CONCEPTS

• **Information systems → Database management system engines**; **Data access methods**; • **Computing methodologies →** *Search methodologies*; *Reinforcement learning*.

KEYWORDS

database performance tuning, reinforcement learning, autonomous database, query optimization

*Corresponding author.

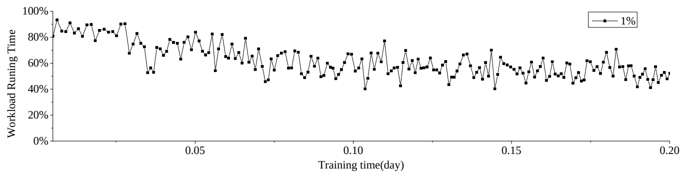
ACM Reference Format:
Jingtian Zhang, Sai Wu, Junbo Zhao, Zhongle Xie, Feifei Li, Yusong Gao, and Gang Chen. 2022. A Sampling-based Learning Framework for Big Databases. In *Proceedings of the ACM Web Conference 2022 (WWW '22), April 25–29, 2022, Virtual Event, Lyon, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3485447.3511991

1 INTRODUCTION

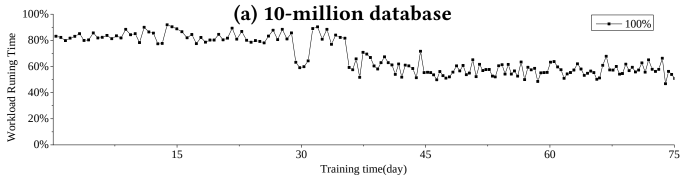
Most, if not all, Web apps are developed upon databases. Alongside the continual rapid booming of the Web, one of its core component in the infrastructure — the database — is urged to evolve concurrently, due to the blast of cumulative information and data. One promising direction for the database of the next generation is to fulfill a complete autonomy; namely through the exploitation of the artificial intelligence technologies, the autonomous databases can automatically resolve a number of crucial fundamental tasks such as query optimization or performance tuning. While these tasks were conventionally accomplished by human-DBAs in the past, it has become increasingly difficult for human intervention due to scalability issues.

One of the most human-intensive tasks in modern database systems — such as the PostgreSQL and an industry database PolarDB — is arguably the database maintenance and tuning task. Notably, more than 90% of DBA's efforts contribute to solving the "slow SQL query" puzzle and sometimes, even experienced DBA crew can fail to address the performance issues for a specific application. Thanks to the rapid advances in machine learning, some recent work [11, 13, 15, 16, 19, 21, 22, 25, 27, 29, 30, 38] pioneers a new research venue, i.e., autonomous database. The concept of an autonomous database is an inherently self-maintained system with little human intervention. A variety of tasks including query optimizations, database configurations and tuning are entirely handled by a continuous learning mechanism.

The majority of the existing work[13, 15, 19, 21, 22, 25, 27, 30, 38] employs a reinforcement-learning based paradigm. The mechanism of such frameworks is to formulate the database tasks as a proxy of markov decision process (MDP). Through extensive exploration and exploitation by interacting with the DMBS, its production is an self-evolving online agent, e.g. predicts the (sub)optimal database



(a) 10-million database



(b) 1-billion database

Figure 1: Index Recommendation Process on TPC-H data with different sizes

configurations or query plans. The performances after the adjustments, normally measured by query latency, query throughput, and storage cost, are fed back as rewards to the MDP to update its built-in parameters by gradient descent. A critical problem in any RL-powered system in recent years is ubiquitously the sample complexity issue; in our setup, to obtain a good agent it may require overly many interactions with the DMBS till convergence. By contrast to the usual game-AI setup, to extensively interact with a DBMS, in particular a big database, is problematic. In particular, overly performing querying or interaction may cause significant computational and time burden in this scenario. More specifically, in Figure 1, we show the training overhead of an index recommendation model proposed in this paper on TPC-H dataset of different sizes (details are presented in the experiment section). The total processing time was employed as our performance metric to train the model. As we can see from the plot, for the small database, we can reduce the running time by half after 2 or 3 hours of training via creating the corresponding indexes. For the large database, to achieve a similar result, the training process lasts for a few days or even months. This indicates that existing reinforcement learning-based index recommendation approaches are too expensive to be applied large-scale databases.

To tackle this problem, in this paper, we propose a transferable sampling-based learning framework with an aim to reduce the potential training overhead for big databases, dubbed as *Mirror*. In particular, *Mirror* consists of a transferring process where the trained policy network is adapted from a sampled database to its original counterpart. While it may seem straightforward, we identify that transferring the policy network between databases of different sizes incur unignorable noises and prediction drifting. In *Mirror*, we further provide a practical sampling algorithm that is guided by rigorous theoretical analysis. To this end, we try to answer the following three questions in *Mirror*.

- What is the difference between the neural model trained in the sampled database and original database?
- How can we guarantee that the model trained in the sampled database is well-adopted to the original database?
- If model transfer incurs a high precision loss, how can we address the problem?

In this paper, we use a typical database tuning task, index recommendation, as an example to illustrate the workflow of *Mirror*. In particular, we build a reinforcement learning framework to support the index recommendation process. Given a target workload consisting of frequent queries, the learning model converges to a final sequence of index actions, minimizing the total query latency.

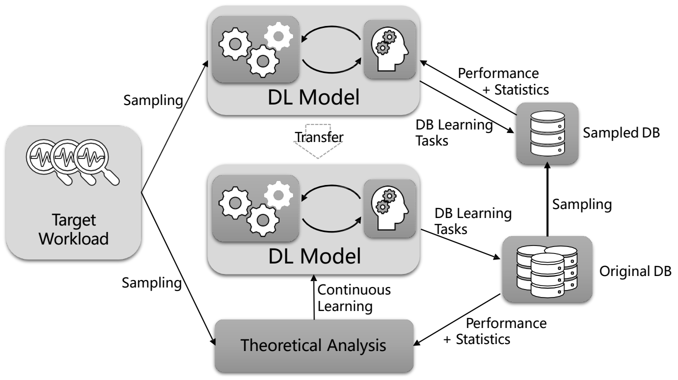


Figure 2: Overview of the *Mirror*

We first train the model on an unbiased sampled database. Then, the *Mirror* framework reuses state-of-the-art results on the robustness of deep neural models, and show how to estimate a proper bound that guarantees that the predicted results work for both databases without affecting the performance. If the bound is violated, *Mirror* applies a continuous learning approach to adjust the result directly on the original database. To reduce the overhead, we only draw necessary training samples from the original database to refine the model. *Mirror* has been integrated into PolarDB as a self-tuning module for our DBA, which currently supports index tuning, cardinality estimation and query plan search. Further, while we showcase the incorporation of *Mirror* into PostgreSQL and PolarDB, *Mirror* can easily be extended to support other database systems.

2 FRAMEWORK OVERVIEW

The general workflow of our framework is shown in Figure 2. *Mirror* works in two phases: the initial training phase and the continuous learning phase. In the initial training phase, *Mirror* first generates an unbiased sampled database, which is two or three orders of magnitude smaller than the original database. Then, a DL (Deep Learning) model is trained for target workload W via exploring the performance of different database configurations on the sampled database. The loss function of the model is to minimize a given performance metric, such as the total processing cost of W.

In our current implementation, we define W as a set of query templates, representing the most popular queries. This strategy is consistent with our application scenario, where PolarDB receives queries submitted from web forms and mobile Apps. Note that, even with the same template, various queries show different processing costs due to varied predicates.

In the second phase, *Mirror* transfers the trained model to the original database and adjusts it based on our theoretical analysis. In particular, a continuous learning approach is employed to refine the model for outliers that cannot be bounded by our theorems. The number of outliers is a trade-off between the accuracy and the performance as the training process on the original database incurs extremely high overheads.

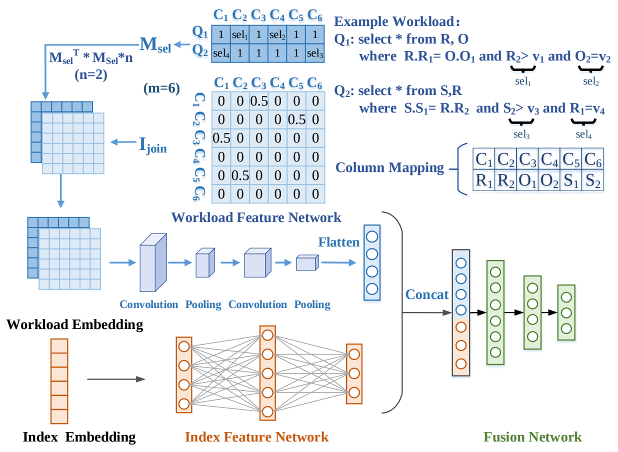
After the second phase, the model is well-tuned for the original database and the target workload. Compared to building a model from scratch, *Mirror* can provide a comparable inference result with an affordable cost for large databases. Moreover, when the root cause of the slow queries is detected[17], *Mirror* can recommend the DBA possible tuning approaches.

In what follows, we introduce the two phases in more details using the index recommendation task as an example. The index recommendation task is defined as:

Definition 1. *For a database D, let C represent its column set. Given a budget λ and target workload W, the recommendation algorithm picks up λ columns for indexing and selects a proper index type(e.g., B-tree, Hash and Bitmap) for each index to minimize the total processing cost for W.*

If there are k types of indexes available for the database, the problem can be simplified as a $(k+1)$-class classification problem. Namely, for each column, we generate a prediction by picking from one of the $(k+1)$ classes. The process repeats λ times, until the top beneficial columns are classified.

2.1 Initial Training Phase



	C_1	C_2	C_3	C_4	C_5	C_6
Q_1	1	sel_1	1	sel_2	1	1
Q_2	sel_4	1	1	1	1	sel_3

	C_1	C_2	C_3	C_4	C_5	C_6
C_1	0	0	0.5	0	0	0
C_2	0	0	0	0	0.5	0
C_3	0.5	0	0	0	0	0
C_4	0	0	0	0	0	0
C_5	0	0.5	0	0	0	0
C_6	0	0	0	0	0	0

Example Workload:
Q_1: select * from R, O
where $R.R_1= O.O_1$ and $\underbrace{R_2> v_1}_{sel_1}$ and $\underbrace{O_2=v_2}_{sel_2}$
Q_2: select * from S,R
where $S.S_1= R.R_2$ and $\underbrace{S_2> v_3}_{sel_3}$ and $\underbrace{R_1=v_4}_{sel_4}$

Column Mapping

C_1	C_2	C_3	C_4	C_5	C_6
R_1	R_2	O_1	O_2	S_1	S_2

Figure 3: Neural Network

In the initial training phase, we build a sampled database using random sampling strategies in [9] and train a DL model for target workload W. To generate training samples, we create a random query set W^q based on our workload template W. The size of W^q, K, is a tunable parameter. Larger K makes the model converge faster but incurs high storage overhead for GPUs. Therefore, our strategy is to pick the largest possible K based on the available GPU memory.

Since *Mirror* modifies its theoretic estimations for each particular model, we briefly introduce our index recommendation model for better clarification. For the index recommendation algorithm task, the architecture of our DL model, shown in Figure 3, consists of several sub-neural networks: workload feature network, index feature network and fusion network.

In the workload feature network, we generate a vector representation for database states (query set W^q and index configuration). Columns from different tables are serialized and assigned a uniform ID as $C_1, C_2, \cdots, C_m$ (see column mapping table in Figure 3). We transform the workload into two $m \times m$ matrices, I_{join} and I_{sel}, which are used to form up the final query embedding I_W.

For the join operators, we consider the equal-join between primary key and foreign key in this paper. Thus, the join information can be represented by an $m \times m$ boolean matrix I_{join}. For column C_i and C_j, their corresponding values ($I_{join}[i,j]$ and $I_{join}[j,i]$) in the matrix denote the probability that $C_i \bowtie C_j$ appears in a query set from the workload. For example, in Figure 3, where two queries, Q_1 and Q_2, are in the workload, the values of two join operators ($R.R_1 = O.O_1$ and $S.S_1 = R.R_2$) are both set to 0.5 since each of them only appears in one query. For other possible join operators, the values are all initialized as 0.

For each query Q_i in a given workload, we use $Sel(Q_i, C_j)$ to denote the selectivity of Q_i's predicate on column C_j, which can be computed by $\frac{|\text{ selected records of } Q_i \text{ on } C_j|}{|\text{ total records of } C_j|}$. If Q_i does not contain any predicate on column C_j, we have $Sel(Q_i, C_j) = 1$. As shown in Figure 3, there are two predicates $R_2 > v_1$ and $O_2 = v_2$ in query Q_1. Therefore, we have $Sel(Q_1, C_2) = sel_1$ and $Sel(Q_1, C_4) = sel_2$. Similarly, we have $Sel(Q_2, C_6) = sel_3$ and $Sel(Q_2, C_1) = sel_4$.

For some columns, the value of $Sel(Q_i, C_j)$ could be less than 0.01%, incurring floating point precision problem. Hence, we use $log(\frac{1}{Sel(Q_i,C_j)})$ instead of $Sel(Q_i, C_j)$ in matrix M_{sel} to ensure the effectiveness of the framework. M_{sel} is a $K \times m$ matrix, where K is the number of queries in W^q and m represents the number of columns in the database schema. To remove the effect of the order of the queries and K, M_{sel} is further transformed into an $m \times m$ dimension matrix $I_{sel} = \frac{1}{n} * (M_{sel}^T \times M_{sel})$.

We use a 2-layer CNN (Convolutional Neural Network) as our workload feature extraction network. A pooling layer is used to compress the output from convolutional layers, which aggregates the features and reduces the computational complexity. The pooling layer also improves the robustness of the extracted features. We use 3×3 convolution kernels and a padding of 1×1 to ensure that the output is the same size as the input, and apply the max-pooling with a stride of 2 to reduce the size by half. Then, the output is flattened as the workload feature vector.

In the index feature network, we use a vector of size $2m$ to indicate whether a column has been indexed or not, namely

$$I_{Index} = [existsIndex(C_1), Type(C_1), \ldots, existsIndex(C_m), Type(C_m)]$$

$existsIndex(C_j)$ is set as 1, if column C_j has been built an index. And $Type(C_1)$ returns the type of the corresponding index (currently, only B-tree, Hash Table and Bitmap are supported). Our index feature network is a 3-layer fully-connected network, where Rectified Linear Unit (ReLU) is used as the activation function.

After the extraction of the workload feature vector and the index feature vector, the two vectors are concatenated and fed into the fusion network. Our fusion network consists of three fully-connected layers and outputs the predicted Q-value vector, which works as an RL (Reinforcement Learning) agent to select the next index building action to maximize the expected reward.

2.2 Continuous Learning Phase

The model trained on the sampled database is transferred to the original database and we apply theoretic bounds to describe the expected prediction of the transferred model (details are shown in Section 3). To further refine the model, we also introduce a continuous learning process to update the model if necessary. The approach is motivated by the following two observations. (1) As mentioned in Section 2, there exist queries for the original database whose selectivity is significantly different from the sampled database. (2) The workload and the data distribution in real systems often change over time. The model cannot predict proper indexes for new queries and new data, and therefore, a model updating strategy is required. In particular, the continuous learning process in *Mirror* includes two steps.

Step 1: Locate the outliers for the transferred model. For a query set W^q, the values of the input matrix I_w to the sampled database and the original database may differ a lot. We consider I_w in the original database as a noisy input for the model trained in the sampled database. Then, we apply our theoretic results on the robustness of the neural model to compute a bound for the noise. Within the bounds, the model returns a consistent result for both databases. Otherwise, the query set W^q is considered as an outlier set. When we obtain enough outlier sets, we start the next step of the continuous learning process.

Step 2: Tune the model with a new branch. To fine-tune the model with outliers, we apply a fusion network. We build a new branch of classification layers (in red rectangle) in the fusion network. The pre-trained parameters of the model are frozen to ensure that learned knowledge on the sampled database is not forgotten. A new layer h_i^{cl} receives the input from both h_{i-1}^{cl} and h_{i-1}. We use V_i and V_i^{cl} to denote the weight matrix of h_i and h_i^{cl}, respectively. The lateral connections from layer $i-1$ of the frozen branch to the layer i of the new branch are denoted as U_i^{cl}. Thus, we have

$$h_i^{cl} = f_{cl}(V_i^{cl} \times h_{i-1}^{cl} + U_i^{cl} \times h_{i-1}),$$

where $f_{cl}(x) = \max(0, x)$. Different from the progressive neural network in [23], we keep only one new branch regardless of the number of outliers to avoid the problem on continuous memory expansion. The architecture of the neural network model is visually illustrated in the appendix.

To speed up the convergence of the model for outlier selection, we apply a prioritized experience replay strategy firstly proposed in [26], which can detect unexpected experience tuples and replay them more frequently during the updating of the neural network. The importance of the experience tuples is measured by the magnitude of a transition's D-error δ, which indicates how unexpected the transition is. That is, how far the value is from its next-step bootstrap estimation [1]. Obviously, the D-errors are larger on the outliers compared to the normal input, which follows similar distributions to the sampled database.

3 ROBUSTNESS OF MODEL TRANSFER

In this section, we justify our approach of transferring the model trained on the sampled database D' to the original database D. In *Mirror*, the DL model can be considered as an agent, which interacts with databases in two ways. First, the model accepts feature embeddings from databases as its input for training and prediction. Specifically, in the index recommendation case, the input includes two matrices, I_{join} and I_{sel}. Second, it asks DBMS to apply the prediction results and test against the target workload to obtain performance metrics as rewards. The rewards are further used as feedbacks to tune the model. In order to produce similar results for both databases, we take the following actions:

- The input from D is considered as a noisy input from D' regarding to the model trained for D'. We show that the effect of noises on the prediction results can be bounded in the rest part of this section.
- The cost of processing a query varies a lot on different sizes of databases and normally does not follow a linear scale-up. We design a neural/cost model approach to adjust the rewards.

Note that the reliability and robustness of DL models is still an open question. Instead of providing rigorous proof, we reuse previous theoretic results to give a best-effort guarantee, which is good enough for most systems.

3.1 Robustness Regarding to Noises

Models in *Mirror* can be considered as multi-class classifiers. For instance, in the index recommendation task, we have : $f : I_{join}^{m\times m} \times I_{sel}^{m\times m} \rightarrow V^m$, where V is a probability vector. The model predicts building an index for the i-th column with a probability $V[i]$. This is a typical multi-class classifier with stacked CNN layers. For simplicity, we further transform it into a two-class classifier as: $f : I_{join}^{m\times m} \times I_{sel}^{m\times m} \times i \rightarrow \{v, 1-v\}$. v is the probability of building index on column i.

In *Mirror*, we first discard the inputs that are not affected by the sampling strategy. In the index recommendation task, I_{join} remains the same for D and D' because it is only determined by the database schema. For the remaining input, we consider them as noises to our trained model. In our example, I_{sel}, computed on D, can be considered as the estimation with noises for D'. To be simple, we use $X = \bar{x} + \delta$ to denote the input to D with a noise δ.

Suppose the correct class for an input $\bar{x}$ on D' is i^* ($i^* = 0$ or 1), we define the margin function for a neural model $f(x)$ as:

$$g(x) = f_{i^*}(x) - f_i(x)$$

$g(x)$ estimates the boundary between the correct class and other classes. If $g(x) \geq 0$, we generate a correct prediction even with noises δ. Otherwise, the model mis-classifies X. Our intuition is to guarantee that most decisions on D and D' are consistent. Therefore, we set a threshold $\epsilon \in (0, 1)$ and try to estimate the probability:

$$Pr(g(x) \geq 0) \geq 1 - \epsilon$$

.

Previous work [34, 37] show that margin function $g(x)$ can be bounded by two linear functions:

$$g^L(x) \leq g(x) \leq g^U(x)$$

where $g^L(x)$ and $g^U(x)$ are defined as:

$$g^L(x) = A^L x + b^L$$

$$g^U(x) = A^U x + b^U$$

A^L and A^U are two constant row vectors and b^L and b^U are two constants. All the parameters can be computed using neural network distillation technique [10]. In this way, if $g^L(x) \geq 0$, we can guarantee that the model is robust against noises.

Reusing the results from PROVEN [33], we have the following theorem for lower and upper bounds (for proof, please refer to the PROVEN paper).

Theorem 1. *Let $f(x)$ be a K-class neural classifier and x_0 is its input. We define the noise δ as $||x - x_0||_p \leq \delta$ for $p \geq 1$. Let $g(x)$ be the margin function. Suppose the input vector X follows some given distribution $\mathcal{D}$ with mean x_0. For a constant $a \geq 0$, there exists a lower bound $\mathcal{L}$ and upper bound $\mathcal{U}$ for the probability $\mathcal{L} \leq Pr(g(x) \geq a) \leq \mathcal{U}$, where*

$$\mathcal{L} = 1 - F_{g^L(x)}(a)$$

and

$$\mathcal{U} = 1 - F_{g^U(x)}(a)$$

$F_Z(z)$ is the cumulative distribution function (CDF) of the random variable Z.

To compute the lower and upper bound, we need to estimate the CDF $F_Z(z)$. In other words, given the distribution of input vector X, we should compute $g^L(x)$ and $g^U(x)$ correspondingly. In *Mirror*, the noises are introduced when performing unbiased sampling for both D and D'. Different from the case of adversarial attacks, the noises in our scenario can be considered as following a multivariate normal distribution with mean x_0 and some covariance Σ.

We define μ_L and μ_U as follows:

$$\mu_L = A^L x_0 + b^L, \mu_U = A^U x_0 + b^U$$

And the variances σ_L and σ_U as:

$$\sigma_L^2 = A^L \Sigma (A^L)^T, \sigma_U^2 = A^U \Sigma (A^U)^T$$

where T denotes the transpose operator.

Note that if X follows a normal distribution with a mean μ and variance Σ, the linear combination $Z = wX + v$ also follows the normal distribution with a mean $\mu_z = w\mu + v$ and variance $\sigma_z = w\Sigma w^T$. The CDF of Z can be estimated as:

$$\frac{1}{2}(1 + erf(\frac{z - \mu_z}{\sigma_z \sqrt{2}}))$$

where erf represents the Gauss error function defined as:

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

Merging the definitions for CDF of Z, μ_L, μ_U, σ_L and σ_U. We have

$$\mathcal{L} \approx \frac{1}{2} - \frac{1}{2} erf(\frac{a - \mu_L}{\sigma_L \sqrt{2}}))$$

$$\mathcal{U} \approx \frac{1}{2} - \frac{1}{2} erf(\frac{a - \mu_U}{\sigma_U \sqrt{2}}))$$

In our current implementation, we evaluate the distance between inputs in L_∞ space. So the noise is computed as $\delta = ||x - x_0||_\infty = argmax_i(|x[i] - x_0[i]|)$. After our model is trained on the sampled database, we can compute the lower bound $\mathcal{L}$ and infer the maximally allowed noise with given confidence (e.g., Pr=95%). In fact, a precise estimation for the bound is not necessary and may be costly. Therefore, we adopt the sampling-based approach from [33], where a set of samples is picked and we apply a bisection search for the noise bound. The bound is gradually refined with new samples.

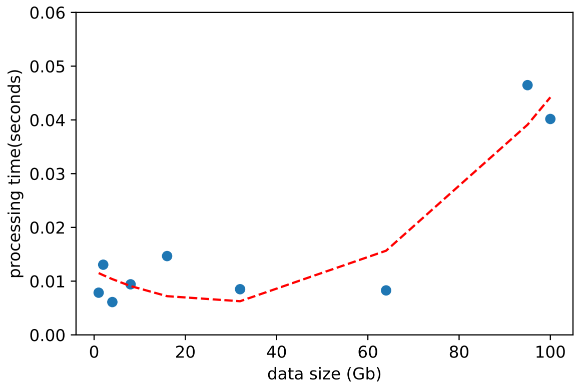


Figure 4: Processing Time on Sampled Databases for Query "select * from customer, orders where customer.c_custkey =orders.o_custkey and c_address='yjUwrTVRDrfVrJ6m0m' and o_orderpriority <'3-MEDIUM' and o_clerk='Clerk#000260479' and c_mktsegment ='HOUSEHOLD';"

The detailed computation of noise on the neural model can be found in Appendix A.

3.2 Normalization of Rewards

For a specific query q_i on a database D, let ρ_i be the sampling ratio of the database. The query cost on the sampled database is represented as $R(q_i, \rho_i)$. Ideally, $R(q_i, \rho_i)$ is a linear function regarding to ρ_i. However, due to incorrect estimation for the statistics and multiple join operations involved, $R(q_i, \rho_i)$ may be difficult to simulate in real systems. Figure 4 shows an example of the processing time for a query on the TPC-H dataset with different sizes. (Other performance metrics show a similar result)

The performance metric is used to compute the reward when training the neural model. To guarantee that the neural model returns the same prediction result for both the original database and any sampled database, we must normalize the rewards on sampled databases so that the prediction results are consistent for all sampled databases. If function R can be learned, the normalization process is defined as:

$$r_{norm}(q_i, s_i) = \frac{R(q_i, 1)}{R(q_i, \rho_i)} * RT(q_i, I)$$

where $RT(q_i, I)$ returns the real processing cost of q_i on sampled database, and $R(q_i, 1)$ denotes the predicted cost of q_i on the original database.

In fact, R is a general cost estimation function for arbitrary queries on sampled databases with varied sizes. It is non-trivial to directly build a model for function R due to the large search space. In this paper, instead of predicting $R(q_i, \rho_i)$, we directly estimate $\frac{R(q_i,1)}{R(q_i,\rho_i)}$. The learning complexity is manageable for:

- $R(q_i, 1)$ and $R(q_i, \rho_i)$ are processing costs for the same query with the same database configurations.

- The statistics estimated from histograms have the same impact on the cost models of the original database and the sampled database[1].
- Given the above two assumptions, both original database and sampled database will process q_i using the same query plan as indicated in Theorem 2.

Theorem 2. *For a database D and its unbiased sampled database D', if both databases adopt the same indexing strategy, they will process a query using the same query plan with a high probability.*

Proof. The proof is in Appendix D. □

To learn the normalized ratio $\frac{R(q_i,1)}{R(q_i,\rho_i)}$, we propose a CAB (Cost model Adjustment using neural Bias) approach. The intuition is that the cost model can provide a baseline estimation for the costs of relational operators. Instead of training an end-to-end neural cost model, we ask the neural network to learn the effect of data sizes on the cost model. The results from our neural model are applied to adjust the prediction of the conventional cost model.

Each unit is designed to predict the ratio of a specific relational operator in a plan. It receives two types of inputs: the hidden state from the previous unit if it has child operators and the input to the original cost model of the database. The input to cost model includes operator type (e.g. "select", "avg" and "join"), column name, table name, predicate and cardinality. The output of the unit includes a hidden state of the current neural bias model and a predicated ratio for this operator. An illustration of the unit structure of our CAB model is provided in Appendix B.

Let H_i, I_0 and I_1 denote the input hidden vector, the inputs to the cost model from the original database and sampled database. The process can be formalized as follows:

$$\begin{aligned} R &= \frac{F(I_0)}{F(I_1)} \\ H_{i+1}, B, W &= G(H_i, V(C_0), V(C_1)) \\ \bar{R} &= W * R + B \end{aligned} \quad (1)$$

F denotes the cost model function of the database. G is the neural bias function and V is a function that vectorizes the input to cost model. B and W are estimated bias and weight, used to normalize the predicted ratio. G is currently implemented as three convolutional layers followed by a ReLU layer. From the above equations, we can observe that we generate an initial estimation using the cost model and then refine it with neural estimation.

To process a query, CAB units are linked together where the hidden state output from one unit is used as input for the next one. Figure 5 shows an example of CAB tree for a query $R \bowtie S$. We generate our prediction for the normalized ratio by leveraging database optimizer. For each operator in the query plan, we create a CAB unit. It receives a hidden state from the CAB of the previous operator in the pipeline and its another input is obtained from the database's cost model and metadata. One special operator is "join", where two hidden states are merged using an average pooling layer. Finally, the root of the CAB tree will output the final predicted result.

[1]The two databases return the same number of unique values and their estimations on the cardinalities are proportional to their sizes. So their cost estimations are proportional to the sizes

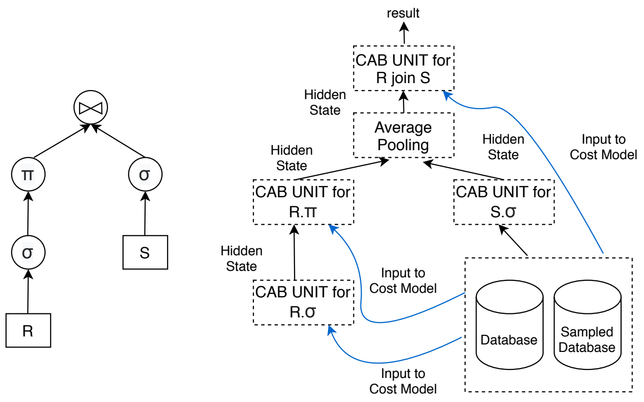


Figure 5: CAB for a Query Plan

One advantage of the CAB is that it has much fewer parameters to learn. It is not required to train the whole CAB tree for each query. Instead, an individual CAB unit is trained. Given a query q and a series of sampled databases $\mathcal{D} = \{D_0, D_1, ...\}$ with different sampling ratios, a series of training pairs can be generated as follows.

First, based on current plan expression tree of q, we can generate a set of sub-queries for q by creating a sub-tree rooted at each inner node, denoted as $Sub(q) = \{\bar{q}_0, \bar{q}_1, ...\}$. For example, we can create three sub-queries for the query in Figure 5. For each sampled database $D_i \in \mathcal{D}$, we execute q on D_i and collect response time (or any other metric), denoted as RT, for each sub-query in $Sub(q)$. Thus, we generate our training set for D_i as:

$$Tr(q, D_i) = \{(\bar{q}_i, RT(\bar{q}_i)) | \forall \bar{q}_i \in Sub(q)\}$$

The detailed training process of the CAB unit is shown in the Appendix.

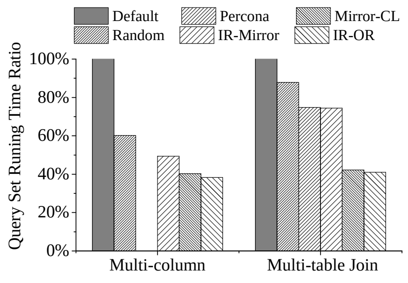
4 EXPERIMENTS

In this section, we conduct extensive experiments to evaluate the performance of *Mirror* on TPC-H benchmark[2], JOB benchmark[3] and the production workload of PolarDB. For TPC-H and JOB benchmark, experiments are conducted on a server equipped with Intel Xeon Processor E5 2660 v2 (25M Cache, 2.20 GHz). For the production workload, we use the pay-as-you-go PolarDB instance equipped with Intel Xeon Platinum 8163 CPU (25M Cache, 2.50GHz). Two database tuning tasks, index recommendation and the cardinality estimation, are tested. The model of cardinality estimation task on *Mirror* is discarded due to limited space. For both tasks, we use V100 GPU to train our model.

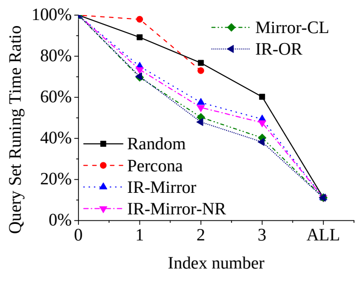
4.1 Index Recommendation Task

We show the performance of our proposed index tuning model on *Mirror*. For comparison, we adapted four baseline approaches in the evaluation. The first approach only builds indexes for all primary keys, denoted as "Default". The second one adopts a random search to select the proper columns for indexing, sometimes a competitive alternative for deep reinforcement learning approach [18, 24]. The

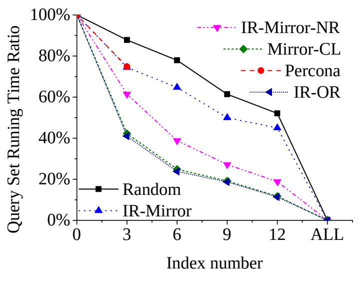
[2]http://www.tpc.org/tpch/
[3]https://github.com/gregrahn/join-order-benchmark



(a) Overall Performance



(b) Index number(Multi-column)



(c) Index number(Multi-table Join)

Figure 6: Result on TPC-H Database

Data Size	IR-Mirror-NR (per V100)	Lift(mins per V100)
8 Million(0.8%)	41 mins	77 mins
16 Million(1.6%)	119 mins	186 mins
32 Million(3.2%)	314 mins	386 mins
1 Billion (100%)	45 days	Not Converge

Table 1: Training Time with Varied Sampling Rate

third one, Percona[4], provides a new indexing approach based on the extensions of PostgreSQL. Finally, we include a recent deep reinforcement learning method following the main idea of [25], denoted as "Lift".

In the experiment, we consider two categories of queries in the evaluation. Namely, "Multi-column Queries" that contain a random number of predicates on a single table and "Multi-table Join Queries" that involve multiple predicates and multiple tables. We use *IR-OR* to denote the method that training our index recommendation model on the original database directly (namely, without *Mirror*), and we denote the model trained on *Mirror* without the optimizations as *IR-Mirror*. We also use *IR-Mirror-NR* and *Mirror-CL* to denote the models trained using reward normalization and continuous learning techniques (*Mirror-CL* includes all proposed optimization techniques in *Mirror*).

By default, all approaches pick the top 3 columns for indexing, because as our experiments show, the performance improvement mainly comes from the first few indexes. We use the workload running time before and after indexing as our metric for each approach, defined as $\frac{t_{index}}{t}$, where t and t_{index} represent the average cost of processing the target workload without/with indexes. t is the same for all approaches.

4.1.1 Performance on TPC-H Database. We generate a TPC-H database with 1 billion tuples. To evaluate the speedup of *Mirror*, we vary the sampling rate of Mirror to generate databases with 8 million, 16 million and 32 million tuples.

Table 1 shows the training time of *IR-Mirror-NR* and *Lift*. For 1 billion database, *IR-Mirror-NR* actually degrades to *IR-OR*, and *Lift* fails to converge with 45 days of training. We do not include the continuous learning time in the table, which is about 1000 minutes on the 1 billion database for all cases. As we can see, even with continuous learning module, *Mirror* still outperforms the training from scratch approach, *IR-OR*.

Then, we evaluate the prediction performance of the proposed sampling-based approach. The result is shown in Figure 6a There is no doubt that *IR-OR* returns the best results, but it only outperforms *Mirror-CL* by a small margin, indicating the effectiveness of our proposed framework. Note that *Lift* is discarded in the diagram, since it cannot converge on the original 1 billion database. However, we report the prediction performance comparison to *Lift* on the small database in Appendix E.

In addition, we show the ability of recommending varied number of indexes. The results are shown in Figure 6b and Figure 6c, where "ALL" indicates that all columns have built an index. We can see that for the first few indexes, *Mirror-CL* achieves the best speedup on the overall performance, which reduces the workload running time to 40% on the Multi-column query set and 42% on the Multi-table Join query set with three indexes, respectively. Due to the index maintenance overhead, for TPC-H database, the optimal index number are 2/3 on the Multi-column query set and 3/6 on the Multi-table Join query set, respectively.

Index number	IR-Mirror	IR-Mirror-NR	Mirror-CL	IR-OR
3	74.4%	61.4%	42.2%	41.0%
6	64.7%	38.8%	24.9%	23.8%
9	49.9%	27.2%	19.2%	18.7%
12	45.0%	18.8%	11.7%	11.4%

Table 2: Ablation Study(Multi-table Join,TPCH)

Last but not the least, we show the ablation study results of our optimization techniques: reward normalization and continuous learning. Table 2 shows the results on the TPC-H database for multi-table join queries. We vary the number of recommended indexes from 3 to 12 and show the query running time after indexing. We observe that both reward normalization and continuous learning techniques can effectively improve the model performances. *Mirror-CL*, where both optimization techniques are applied, achieves a similar result as the *IR-OR* which trains a model directly on the original big database. However, even equipped with the continuous learning process, the training time is an order of magnitude smaller than the training time on the original database.

4.1.2 Performance on PolarDB Instance. In this section, we evaluate the proposed sampling-based learning method on the pay-as-you-go PolarDB instance. Similar to previous evaluation, we include *IR-Mirror*, *Mirror-CL* and *IR-OR* approaches for comparisons.

[4]https://www.percona.com/blog/2019/07/22/automatic-index-recommendations-in-postgresql-using-pg_qualstats-and-hypopg/

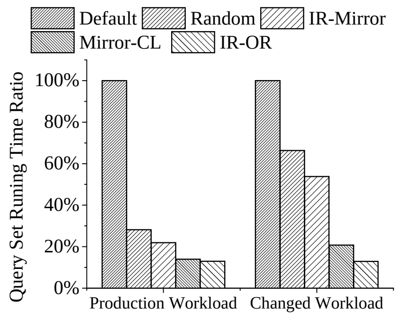


Figure 7: Overall Performance(PolarDB)

First, we collect 7-day's cluster trace data from an online PolarDB setup. We set the sampling ratio of *Mirror* as 1% and show performances of the recommended indexes. Then, to simulate the real scenario, where distributions of workload change over time, we insert the trace data on the 8th day and conduct the continuous learning for *Mirror-CL*. For the changed workload, we use *Mirror-CL* and *IR-Mirror* to denote the model transferred with and without continuous learning respectively. The results are shown in Figure 7, indicating that our framework can not only achieve a good prediction result on the production workload, but also handle gradually updated query distributions effectively.

4.2 Cardinality Estimation Task

Cardinality estimation is another important database learning task, which estimates the number of resulted tuples after a series of relational operators. The success of a database query optimizer relies on a precise cardinality estimation model. In this section, we show the performance of *Mirror* for the cardinality estimation model. Similar to the previous evaluation, our sampling-based method is denoted as "Mirror-CL" We use the widely adopted JOB benchmark[5] (21 tables and the largest table has 36 million rows) in the evaluation.

We use three existing sampling-based cardinality estimation method as baselines, namely, "CS2"[36], "CS2l"[4] and "CSDL"[32]. We also include the cardinality estimation result from PostgreSQL's optimizer, denoted as "PG" in the diagram. In the test, we adopt the q-error metric, defined as $max(Card_q/Card'_q, Card_q/Card'_q)$, where $Card_q$ and $Card'_q$ are the extract number and the estimated number of records satisfying all predicates in the query q respectively. Similar to previous work, we report the q-error distribution (50%, 90%, 95%,99% and 100%) of each query workload. Table 3 reports the q-error distribution for different algorithms. We can see that the q-error of all the sampling methods is less than 10 for 90% of all queries, which means they accurately estimate the magnitude of the cardinality value for most queries. However, "Mirror-CL" outperforms "PG" by 87X, "CS2" by 73X, "CS2l" by 2.3X and "CSDL" by 1.2X at the 95%-quantile. For the upper ranges, the improvement on the second best sampling method is around 3.

5 RELATED WORK

Deep learning approaches have been widely applied to improve the database capabilities, such as cost estimation [11, 16], query plan optimization [13, 19, 21, 30] and database tuning [15, 22, 29, 38]. A detailed survey on recent advance can be found in [39]. Most existing work adopt the reinforcement learning models to interact with databases [13, 15, 19, 21, 22, 30, 38]. However, these work uniformly only focus on small-to-medium sized databases due to the computational overhead, unlike *Mirror*.

[5]https://github.com/gregrahn/join-order-benchmark

Algorithm	50%	90%	95%	99%	Max
PG	3.37	48.9	101	781	1.23×10^3
CS2	1.16	1.42	85.3	1.14×10^3	1.38×10^5
CS2L	1.08	1.23	2.7	27.1	45
CSDL	1.11	1.17	1.39	3.85	7.1
Mirror-CL	1.02	1.13	1.17	1.2	2.13

Table 3: Q-Error of cardinality estimation algorithms on Multi-table Join.

Around the index recommendation task, the previous work, such as [3, 7, 8, 14] leveraging the database cost models, may not produce an optimal index strategy [14]. To address this problem, learning-based techniques are proposed. Azure takes automated corrective actions (e.g., automatically reverting the created index) to fix 11% query performance regressions [5] and applies a neural network to compare the workload cost under different index configurations instead of the what-if caller [6]. Predictive indexing [2] refines the model using feedbacks in the next recommendation cycle. Sun et al. [28] proposed an end-to-end cost estimator to support index recommendation. NoDBA [27] proposed a deep Reinforcement Learning controller for index selection, where workloads are assumed to be single-tabled on a small database. LIFT [25] constructed a controller using Double Q-learning model [31] to solve the index recommendation problem on small document databases. Thus, neither of them can be applied to huge in-production databases.

An orthogonal line of research is the index structure search. The learned index family [12, 20, 35] is recently proposed to replace the traditional index structures and has shown superior performance for multiple scenarios. Our framework can be combined with them as a full-fledged index tuning module.

6 CONCLUSIONS

In this paper, we propose *Mirror* to support tuning tasks on big databases. The intuition of *Mirror* is to reduce the training overhead by transferring a trained policy network. *Mirror* theoretically analyzes when the DL model can be transferred to the original database for a given workload. If the theoretic bounds are violated, *Mirror* adopts a continuous learning technique to refine the model on the original database. Experiments demonstrate promising results. In the future, we plan to implement more database learning tasks on the *Mirror*.

ACKNOWLEDGMENTS

This work is funded by the Fundamental Research Funds for Alibaba Group through Alibaba Innovative Research (AIR) Program, the NSFC [grant numbers 61872315] and the Zhejiang Provincial Natural Science Foundation [grant number LZ21F020007]. This work is also sponsored by the Fundamental Research Funds for the Central Universities of China.

REFERENCES

[1] David Andre, Nir Friedman, and Ronald Parr. 1997. Generalized Prioritized Sweeping. In *NIPS*. 1001–1007.

[2] Joy Arulraj, Ran Xian, Lin Ma, and Andrew Pavlo. 2019. Predictive Indexing. *CoRR* abs/1901.07064 (2019).

[3] Renata Borovica, Ioannis Alagiannis, and Anastasia Ailamaki. 2012. Automated physical designers: what you see is (not) what you get. In *Proceedings of the Fifth International Workshop on Testing Database Systems, DBTest 2012, Scottsdale, AZ, USA, May 21, 2012*. 9.

[4] Yu Chen and Ke Yi. 2017. Two-Level Sampling for Join Size Estimation. In *SIGMOD*. 759–774.

[5] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *SIGMOD*. 666–679.

[6] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *SIGMOD*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1241–1258.

[7] Bailu Ding, Sudipto Das, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2018. Plan Stitch: Harnessing the Best of Many Plans. *PVLDB* 11, 10 (2018), 1123–1136.

[8] Adam Dziedzic, Jingjing Wang, Sudipto Das, Bolin Ding, Vivek R. Narasayya, and Manoj Syamala. 2018. Columnstore and B+ tree - Are Hybrid Physical Designs Important?. In *SIGMOD*. 177–190.

[9] Peter J. Haas and Joseph M. Hellerstein. 1999. Ripple Joins for Online Aggregation. In *SIGMOD*. 287–298.

[10] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. 2015. Distilling the Knowledge in a Neural Network. *CoRR* abs/1503.02531 (2015). arXiv:1503.02531 http://arxiv.org/abs/1503.02531

[11] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*.

[12] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 489–504.

[13] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. *CoRR* abs/1808.03196 (2018).

[14] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2020. An Index Advisor Using Deep Reinforcement Learning. In *CIKM*, Mathieu d'Aquin, Stefan Dietze, Claudia Hauff, Edward Curry, and Philippe Cudré-Mauroux (Eds.). ACM, 2105–2108.

[15] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *PVLDB* 12, 12 (2019), 2118–2130.

[16] Henry Liu, Mingbin Xu, Ziting Yu, Vincent Corvinelli, and Calisto Zuzarte. 2015. Cardinality estimation using neural networks. In *CASCON*. 53–59.

[17] Minghua Ma, Zheng Yin, Shenglin Zhang, Sheng Wang, Christopher Zheng, Xinhao Jiang, Hanwen Hu, Cheng Luo, Yilin Li, Nengjun Qiu, Feifei Li, Changcheng Chen, and Dan Pei. 2020. Diagnosing Root Causes of Intermittent Slow Queries in Large-Scale Cloud Databases. *Proc. VLDB Endow.* 13, 8 (2020), 1176–1189.

[18] Horia Mania, Aurelia Guy, and Benjamin Recht. 2018. Simple random search provides a competitive approach to reinforcement learning. *CoRR* abs/1803.07055 (2018).

[19] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *SIGMOD*. 3:1–3:4.

[20] Ryan Marcus, Emily Zhang, and Tim Kraska. 2020. CDFShop: Exploring and Optimizing Learned Index Structures. In *SIGMOD*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2789–2792.

[21] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2018. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In *SIGMOD*. 4:1–4:4.

[22] Cheng Peng, Canqing Zhang, Cheng Peng, and Junfeng Man. 2017. A reinforcement learning approach to map reduce auto-configuration under networked environment. *IJSN* 12, 3 (2017), 135–140.

[23] Andrei A. Rusu, Neil C. Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. 2016. Progressive Neural Networks. *CoRR* abs/1606.04671 (2016).

[24] Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. 2017. Evolution Strategies as a Scalable Alternative to Reinforcement Learning. *CoRR* abs/1703.03864 (2017).

[25] Michael Schaarschmidt, Alexander Kuhnle, Ben Ellis, Kai Fricke, Felix Gessert, and Eiko Yoneki. 2018. LIFT: Reinforcement Learning in Computer Systems by Learning From Demonstrations. *CoRR* abs/1808.07903 (2018).

[26] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2016. Prioritized Experience Replay. In *ICLR*.

[27] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. The Case for Automatic Database Administration using Deep Reinforcement Learning. *CoRR* abs/1801.05643 (2018).

[28] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *CoRR* abs/1906.02560 (2019).

[29] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. 2019. iBTune: Individualized Buffer Tuning for Large-scale Cloud Databases. *Proc. VLDB Endow.* 12, 10 (2019), 1221–1234.

[30] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. In *SIGMOD*. 1153–1170.

[31] Hado van Hasselt, Arthur Guez, and David Silver. 2016. Deep Reinforcement Learning with Double Q-Learning. In *AAAI*. 2094–2100.

[32] TaiNing Wang and Chee-Yong Chan. 2020. Improved Correlated Sampling for Join Size Estimation. In *ICDE*. 325–336.

[33] Tsui-Wei Weng, Pin-Yu Chen, Lam M. Nguyen, Mark S. Squillante, Ivan V. Oseledets, and Luca Daniel. 2018. PROVEN: Certifying Robustness of Neural Networks with a Probabilistic Approach. *CoRR* abs/1812.08329 (2018).

[34] Tsui-Wei Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane S. Boning, and Inderjit S. Dhillon. 2018. Towards Fast Computation of Certified Robustness for ReLU Networks. In *ICML (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer G. Dy and Andreas Krause (Eds.). PMLR, 5273–5282.

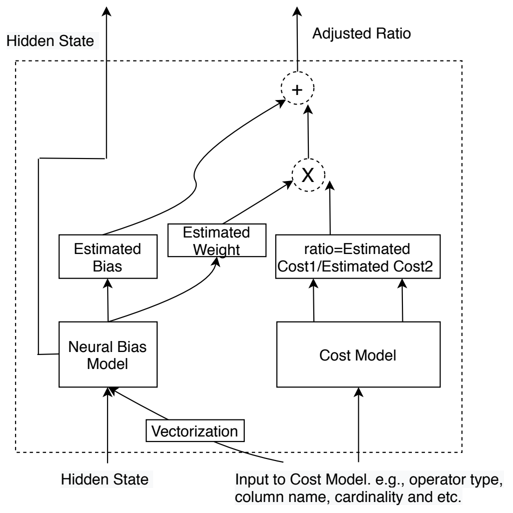
[35] Sai Wu, Xinyi Yu, Gang Chen, Yusong Gao, Xiaojie Feng, and Wei Cao. 2019. Progressive Neural Index Search for Database System. *CoRR* abs/1912.07001 (2019).

[36] Feng Yu, Wen-Chi Hou, Cheng Luo, Dunren Che, and Mengxia Zhu. 2013. CS2: a new database synopsis for query estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. 469–480.

[37] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. 2018. Efficient Neural Network Robustness Certification with General Activation Functions. *CoRR* abs/1811.00866 (2018). arXiv:1811.00866 http://arxiv.org/abs/1811.00866

[38] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *SIGMOD*. 415–432.

[39] X. Zhou, C. Chai, G. Li, and J. SUN. 2020. Database Meets Artificial Intelligence: A Survey. *IEEE Transactions on Knowledge and Data Engineering* (2020), 1–1. https://doi.org/10.1109/TKDE.2020.2994641



Hidden State
Adjusted Ratio
+
X
Estimated Bias
Estimated Weight
ratio=Estimated Cost1/Estimated Cost2
Neural Bias Model
Cost Model
Vectorization
Hidden State
Input to Cost Model. e.g., operator type, column name, cardinality and etc.

Figure 8: Unit Structure of CAB

A NOISE COMPUTATION

According to the theoretical analysis in Section3.1, we can compute the noise as

$$\delta = argmax_{i=0}^{m} |\sum_j \log(\frac{1}{Sel(Q_j, C_i)})^2 - \sum_j \log(\frac{1}{Sel'(Q_j, C_i)})^2|$$

where Sel and Sel' return different selectivity estimations for D and D'.

The below table shows the noise bound for different confidences ϵ on our neural model, where values of input vector are normalized within [0, 1]. $\epsilon = 100\%$ indicates that no false classification is allowed.

$\epsilon = 100\%$	$\epsilon = 99\%$	$\epsilon = 90\%$	$\epsilon = 85\%$	$\epsilon = 80\%$
0.00812	0.0220	0.0232	0.0232	0.0323

Table 4: Maximal Noise Bound with RELU function

B SOME VISUAL ILLUSTRATIONS

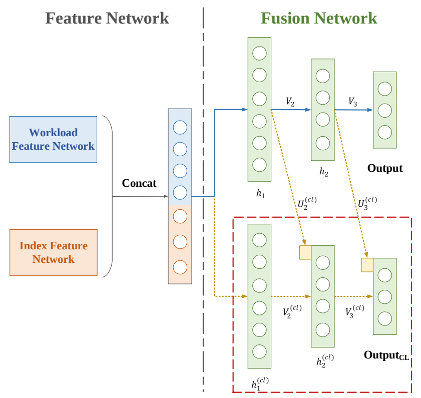


Figure 9: Neural network for Continuous Learning

In Figure 9, we display the architecture of the neural network devised for the continuous learning phase. Besides, Figure 8 shows the unit structure of our CAB model.

```
Input: Database $D$, Workload $Q$
Output: CAB model
1  for $i$ = 1 to $k$ do
2      $D_i$ = unbiased samples of $D$ with a random ratio for $\forall q_j \in Q$ do
3          $Sub(q_j)$ = getSubQueries($q_j$)
4          $Tr(q_j, D_i)$ = createTrainingPairs($Sub(q_j)$, $D_i$)
5      end
6  end
7  for $\forall q_i \in Q$ do
8      for $\forall q_j \in Sub(q_i)$ do
9          $tree$ = CreateCABTree($q_j$)
10         pick two random sampled database $D_0$ and $D_1$
11         Train $tree$ with $Tr(q_j, D_0)$ and $Tr(q_j, D_1)$
12     end
13 end
```

Algorithm 1: Training of CAB Unit

C TRAINING THE CAB MODEL

The training process is summarized in Algorithm 1. During the training process, we construct the CAB tree based on the query plan returned by the query optimizer of database for the input query. To reduce the training overhead, all CAB units share the same neural parameters and hence.

D PROOF OF THEOREM 2

.

c_{sp}	Cost of a sequential page fetch
n_{sp}	Number of sequential page fetches
c_{rp}	Cost of a random page fetch
n_{rp}	Number of random page fetches
c_{ip}	Cost of accessing an index entry
n_{ip}	Number of probed index entries
c_{op}	Average cost of CPU per operator
n_{op}	Number of operations involved

Table 5: Table to test captions and labels

THEOREM 3. *Theorem 2 restated. For a database D and its unbiased sampled database D′, if both databases adopt the same indexing strategy, they will process a query using the same query plan with a high probability.*

PROOF. We use PostgreSQL as an example[6]. Using the parameters defined in Table 5, the database's cost model estimates the cost C_{op_k} of an operator op_k in a query using a linear combination:

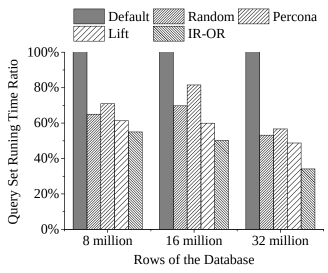
$$C_{op_k} = c_{sp} \cdot n_{sp} + c_{rp} \cdot n_{rp} + c_{tp} \cdot n_{tp} + c_{ip} \cdot n_{ip} + c_{op} \cdot n_{op} \quad (2)$$

For a given query plan P consisting of different operators, the cost of P is the sum of the costs of all operators:

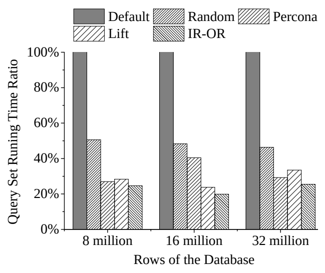
$$C_P = \sum_{op_k \in P} C_{op_k} \quad (3)$$

D and D' will adopt the same values for all c_x parameters. If we use histograms to estimate n_x, we find have $\rho = \frac{D'.c_x}{D.c_x}$, where ρ is the sampling ratio. In this way, if a plan P is the optimal for D, it must also be the optimal one for D'. Otherwise, we can also find a new optimal plan for D. □

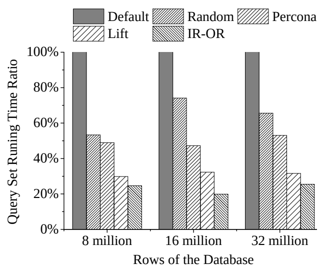
[6]Our analysis method can be also applied to other SQL databases due to the high similarity of the cost functions in the systems.



(a) Single-column



(b) Multi-column



(c) Multi-table Join

Figure 10: Overall Performance(TPC-H)

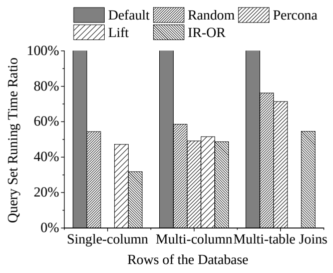
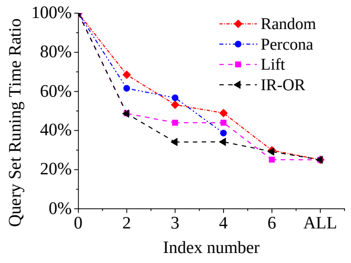
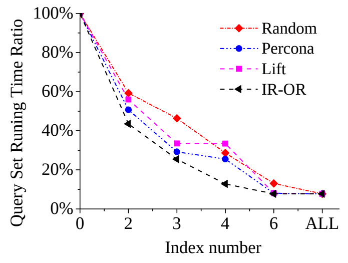


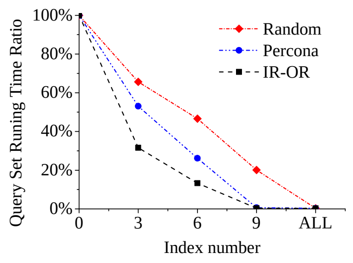
Figure 11: Overall Performance(JOB)



(a) Single-column

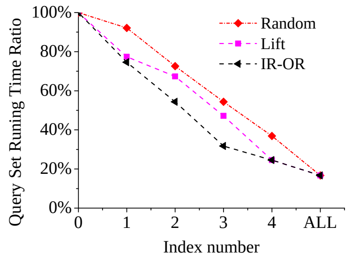


(b) Multi-column

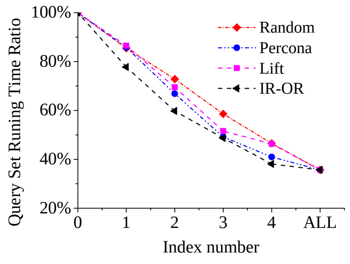


(c) Multi-table Join

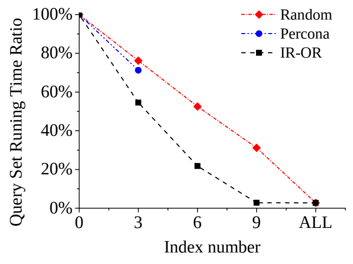
Figure 12: Index number(32 Million), TPC-H



(a) Single-column



(b) Multi-column



(c) Multi-table Join

Figure 13: Index number, JOB

E EXPERIMENTS ON THE INDEX RECOMMENDATION MODEL

We have conducted extensive experiment study on the index recommendation task. In addition to the TPC-H benchmark, we consider the JOB[7], whose databases are generated based on the real IMDB dataset, in the performance evaluation too. Besides, we also include the Single-column Queries that only involve one predicate and one table in the experiment. For each query type, the experiments are conducted on the testing workloads where each workload contains 50 randomly generated queries with predefined templates. For each workload, we use the query templates of shape $SELECT * FROM \cdots WHERE \cdots$.

In the first test, we vary the sampling ratio with 8 million(0.8%), 16 million(1.6%) and 32 million(3.2%) records, respectively. The results on the TPC-H benchmark are shown in Figure 10. All approaches recommend top three beneficial indexes and we compare the percentage of improvement on average running time before/after indexing. It is no doubt that *IR-OR* performs the best, especially for the complex case involving multiple predicates and multiple joins. Note that "Lift" is discarded in Figure 10c, as it does not support joins. The results on the JOB benchmark are shown in Figure 11. We fix the size of database and the sampling ratio is set as 10%. Because *Percona* does not return any valid result and *Lift* cannot be applied to multi-table scenarios, we omit them correspondingly.

We also explore the effect of varied number of indexes. For the TPC-H benchmark, we adopt the sampled database with 32 million rows. As the figure shows, our approach can achieve the optimal result compared to other baselines, significantly reducing the query processing overhead. We also observe that the improvement of indexes decreases when a large number of indexes have already been built. Similarly, we show the results on the real IMDB dataset in Figure 13, where *IR-OR* performs the best in all cases.

[7]https://github.com/gregrahn/join-order-benchmark