Learning-based Sample Tuning for Approximate Query Processing in Interactive Data Exploration

Hanbing Zhang, Yinan Jing *Member, IEEE*, Zhenying He, Kai Zhang, and X. Sean Wang *Senior Member, IEEE*

***Abstract*—For interactive data exploration, approximate query processing (AQP) is a useful approach that usually uses samples to provide a timely response for queries by trading query accuracy. Existing AQP systems often materialize samples in the memory for reuse to speed up query processing. How to tune the samples according to the workload is one of the key problems in AQP. However, since the data exploration workload is so complex that it cannot be accurately predicted, existing sample tuning approaches cannot adapt to the changing workload very well. To address this problem, this paper proposes a deep reinforcement learning-based sample tuner, *RL-STuner*. When tuning samples, *RL-STuner* considers the workload changes from a global perspective and uses a Deep Q-learning Network (DQN) model to select an optimal sample set that has the maximum utility for the current workload. In addition, this paper proposes a set of optimization mechanisms to reduce the sample tuning cost. Experimental results on both real-world and synthetic datasets show that *RL-STuner* outperforms the existing sample tuning approaches and achieves 1.6×-5.2× improvements on query accuracy with a low tuning cost.**

***Index Terms*—Approximate query processing, Interactive data exploration, Data analysis.**

I. Introduction

TODAY, *Interactive Data Exploration* (IDE) is widely used to help users get insights and inspirations from a large volume of data. IDE is a typical "*human-in-the-loop*" application, where users incrementally refine filters and visualize subsets of data using multiple plots for different attributes and aggregate functions [1], [2]. The workload in IDE is typically composed of sessional queries and ad-hoc queries. Sessional queries are usually several sequential and iterative queries that are inter-dependent [2]. When the user's interest changes, an ad-hoc query that is independent with previous sessional queries will be posed to explore data through different attributes, such as independent browsing defined in [2]. These two types of queries are usually interwoven together in the workload of IDE. Hence, the workload in IDE is much more complex than the static reporting-style analytical workload and is continuously changing along with the changes of the user's interests.

In IDE tasks, answering queries in a short response time is very important to user experience, because high query latency will adversely inhibit users' productivity and creativity [3], [4]. Approximate query processing (AQP) is one of the promising techniques to meet the requirements of timely response in data exploration by trading query accuracy for query response time [5]. Since data exploration usually relies just on a "big picture", approximate results with an accuracy guarantee are acceptable. The sampling-based AQP techniques [6]–[12] use small samples instead of the whole underlying dataset to answer queries. There are two ways to generate samples for AQP. One is online sampling that generates samples at runtime for a specific query. For example, the online AQP engine Quickr [9] optimizes the query execution by incorporating sampler operators into a cost-based query optimizer and generating samples for a given query at runtime. However, online sampling is usually expensive and benefits only one specific query. If we can reuse these samples generated online, the sampling cost can be amortized in the future queries. Another is offline sampling that pre-computes samples from the underlying data according to the query workload that is known a priori. For example, the well-known offline AQP engine BlinkDB [7] pre-computes stratified samples on the most frequently used columns that appeared in the query workload while keeping the total storage costs below the given storage budget. Compared with the expensive online sampling, offline sampling can help AQP systems significantly reduce the query latency by leveraging on the pre-computed samples. However, the need for a priori knowledge of the workload makes that offline sampling cannot adapt to the workload changes in IDE.

Taster [13] combines the benefits of both online and offline sampling. It materializes samples generated online for reuse in future queries and continuously tunes the samples from a local perspective to adapt to the workload changes. When tuning samples, Taster assumes the recently appeared queries are a good representation of future queries and uses a greedy algorithm [13], [14] to approximate the optimal sample set which has the maximum utility for these recent queries. To the best of our knowledge, this is the state-of-the-art sample tuning approach in AQP systems. However, the assumption in Taster cannot always hold for all future queries in IDE, and the greedy algorithm in Taster cannot select the optimal sample set. Therefore, Taster might fail to select appropriate samples that can bring the maximum utility for the current workload to materialize. Furthermore, since there may be an overlap between the samples in the AQP systems, the existing page replacement algorithms for operating system (e.g., FIFO and LRU) are difficult to use to solve the sample tuning problem. In summary, for AQP systems, how to tune samples to materialize

Hanbing Zhang, Yinan Jing, Zhenying He, Kai Zhang, and X. Sean Wang are with the School of Computer Science, Fudan University, Shanghai, China.
Corresponding Author: Yinan Jing
E-mail: jingyn@fudan.edu.cn

Manuscript received xx; revised xx; accepted xx. Date of publication xx; date of current version xx. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. xx

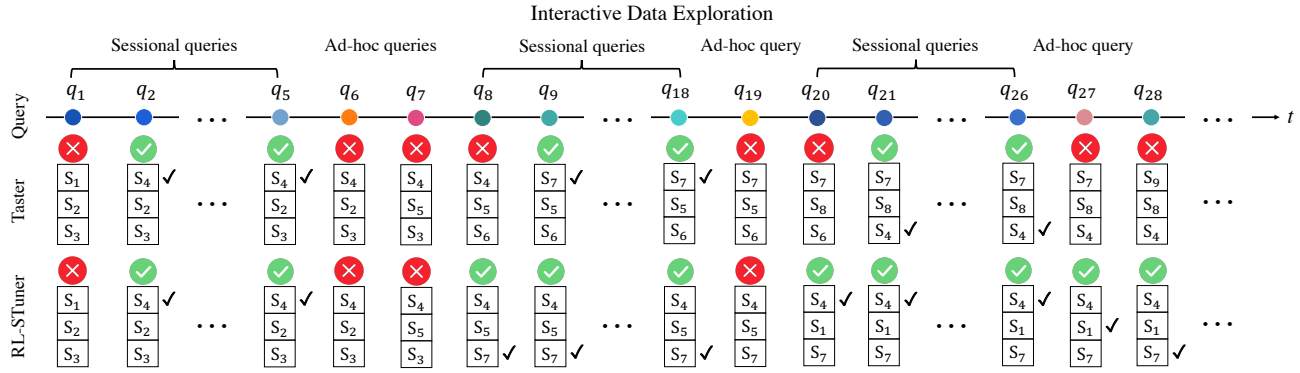


Fig. 1: An example of interactive data exploration to show how Taster and *RL-STuner* tune samples to adapt to the workload changes. q_i represents a query in data exploration. The queries with the similar color are inter-dependent with each other. Assume the memory fits three samples and S_j is a sample. Marks ✔ and ✘ illustrate whether the current query can be answered by existing samples. The sample with ✓ represents it is used to answer the current query.

according to the changing workload in IDE is a challenging and fundamental problem.

In this paper, to address the sample tuning problem in AQP, we propose a learning-based sample tuner, called *RL-STuner*, which tunes samples from a global perspective by using a deep reinforcement learning model. When tuning samples, *RL-STuner* learns from all appeared queries and leverages this global knowledge to guide sample tuning. To avoid the heavy overhead for computing the optimal sample set from a large number of possible combinations of samples, *RL-STuner* uses a Deep Q-learning Network (DQN) model [15] to get an approximate optimal solution, which stores the feedback of tuning samples in different episodes and uses them to guide the sample selection in one sample tuning. For example[1], Figure 1 shows an interactive data exploration and how Taster and *RL-STuner* tune samples to adapt to the workload changes. As shown in Figure 1, the workload is composed of several sessional queries and ad-hoc queries. To demonstrate the inter-dependent feature of queries in the same session, we use the circles filled with similar colors to represent a session of queries. For simplicity, we assume the memory fits three samples. If one of the existing samples can be used to answer the current query, we mark ✔. Otherwise, we mark ✘ and it will trigger a round of sample tuning. As shown in Figure 1, overall, *RL-STuner* outperforms Taster, since *RL-STuner* tunes the samples from a global perspective, while Taster only leverages the recent w (assuming $w = 3$) queries to guide tuning. For instance, *RL-STuner* can support q_8 because it learns knowledge from all previous queries ($q_1 \sim q_7$) and recommends S_7 that can be used to answer q_8, while Taster just consider the recent 3 queries ($q_5 \sim q_7$). As for q_{20} and q_{28}, they also benefit from the global perspective tuning of *RL-STuner*. As for q_{27}, although it is an ad-hoc query, *RL-STuner* still can support this query because it can learn knowledge from previous simialr ad-hoc queries $\{q_6, q_7, q_{19}\}$.

Furthermore, since the workload in IDE is changed dynamically, it is best to tune the samples as soon as possible. Otherwise, the stale samples cannot support the AQP system to return the query result with high accuracy for the changing workload. However, tuning the samples by recomputation from scratch will incur a heavy overhead once the workload changes. The expensive sample tuning cost will adversely inhibit the performance of AQP systems. To mitigate this problem, we propose three optimization mechanisms to reduce the sample tuning cost in *RL-STuner*. First, we propose a lazy sample tuning strategy to reduce the sample selection and tuning operations during tuning. Second, we use a parameter transfer approach to initialize the DQN model in the sample tuning task with a suitable solution to reduce the training cost. Third, we propose a utility estimator to avoid accessing the underlying data when calculating the utility of samples to reduce the calculation cost. By taking these three optimization mechanisms, *RL-STuner* can quickly tune the samples to adapt to the changing workload.

To sum up, this paper makes the following contributions:

- We propose a deep reinforcement learning-based sample tuner, *RL-STuner*, which tunes samples from a global perspective by using a DQN model, to make the AQP system be able to adapt to the changing workload in IDE.
- We propose a set of optimization mechanisms to systematically reduce the sample tuning cost in *RL-STuner*: a lazy sample tuning strategy to avoid unnecessary sample selection and tuning operations, a parameter transfer approach to reduce the training cost of the DQN model, and a utility estimator to reduce the computational cost of sample utility.
- We intensively evaluate *RL-STuner* under various workloads with the state-of-the-art IDE benchmark [2]. Experimental results on both real-world and synthetic datasets show that *RL-STuner* outperforms the existing sample tuning approaches and achieves 1.6×-5.2× improvements on query accuracy of AQP queries in IDE.

II. Framework and Preliminaries

A. Framework

Figure 2 shows an AQP system framework, into which we incorporate a sample tuner. The system is composed of

[1]The workload used in this example can be retrieved from https://github.com/DogeWang/RL-STuner.

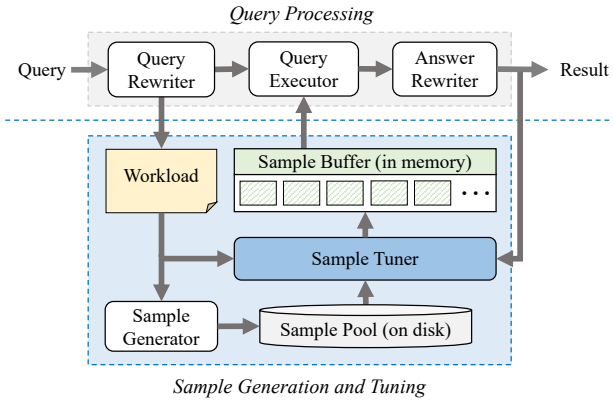


Fig. 2: An AQP system with a sample tuner

three parts: *Query Processing*, *Sample Generation* and *Sample Tuning*.

Query Processing: This part is responsible for answering AQP queries by using samples. At first, a query will be rewritten as an AQP query on samples. When executing this query, the system will try to find a suitable sample from the *sample buffer* that is a memory buffer for storing samples. **Specifically, if the selection conditions of a new query are covered by the selection conditions of a previous query, we can reuse the sample generated according to the previous query to answer the new query since the sampling range of this sample can cover the new query.** Note that if more than one sample in the sample buffer can satisfy the query, we will choose the sample with the largest sampling ratio to answer this query because such a sample can usually provide a higher query accuracy. Finally, the system will return an approximate result to the user.

Sample Generation: The sample generator is responsible for generating samples from the underlying data according to the incoming queries. **Note that the sample generation does not require prior knowledge about the workload. Initially, when no query arrives, the sample generator first generates a uniform sample from the underlying data. Then, when a query is posed, the sample generator generates samples according to the characteristics of that query. Specifically, if there are GROUP BY clauses in the posed query, the sample generator will generate stratified samples based on the attributes in the GROUP BY clauses. Otherwise, the sample generator will generate uniform samples for this query. If there are WHERE clauses in the posed query, the sample generator will generate one or multiple samples from data subsets according to the selection conditions in the WHERE clauses. For the stratified sampling, the size of the sample in each stratum is $|S|/N_s$, where $|S|$ is the sample size (i.e., sampling ratio multiplied by the total size of underlying data) and N_s is the number of strata. Note that if a specific sample has been generated according to the previous queries, the sample generator will not generate the same sample again.** The metadata of each sample is stored in the memory. Thanks to the cheap disk price, the sample pool can usually maintain all the samples that have been generated.

Sample Tuning: In order to reduce the cost of accessing samples and speed up query processing, we use a sample buffer in memory to maintain the frequent-used samples. Due to the limited memory space, obviously, the sample buffer cannot store all samples. Therefore, we need to select a subset of samples from the sample pool to maximize the utility of samples. In our system, the sample tuner is responsible for this task. Once the system cannot find a suitable sample from the sample buffer to answer the query or the accuracy of the result cannot satisfy the user's accuracy requirement, the offline sample tuning will be triggered. In this paper, to adapt to the changing workload in IDE, we propose a deep reinforcement learning-based sample tuner (*RL-STuner*). **When a sample tuning is triggered, *RL-STuner* tunes the sample buffer by using a DQN model to select an optimal sample set with the maximum utility according to the queries that users have posed. Note that the DQN model in *RL-STuner* will only learn historical queries that have appeared before this tuning and will not learn anything about future queries.** During the tuning, if a sample is obsoleted from the sample buffer by the sample tuner, it will be moved back to the sample pool to avoid repeated sample generation.

B. Preliminaries

Sample and Active sample. In this paper, we denote all the generated samples as $\mathbb{S} = \{S_1, S_2, \cdots, S_n\}$, which are stored in the sample pool. Since the samples in the sample buffer can be used directly to answer the query, we call these samples stored in the sample buffer as *active samples*, denoted as $\widetilde{\mathbb{S}}$ ($\widetilde{\mathbb{S}} \subseteq \mathbb{S}$), which are selected from $\mathbb{S}$ according to a specific sample tuning strategy by the sample tuner.

Benefit of an active sample. For each active sample $\widetilde{S}$ ($\widetilde{S} \in \widetilde{\mathbb{S}}$), if it can be used to answer a specific query q (denoted as $\widetilde{S} \rightarrow q$), the benefit of using $\widetilde{S}$ to answer q is the weighted summation of the improvements on query latency and query accuracy. The improvement on query latency $RL(q|\widetilde{S})$ is calculated by $RL(q|\widetilde{S}) = (L(q|D) - L(q|\widetilde{S}))/L(q|D)$, where $L(q|D)$ is the query latency of using D to answer q, and $L(q|\widetilde{S})$ is the query latency by using $\widetilde{S}$ to answer q. For the improvement on query accuracy, we use the negative value of the relative error (RE) to evaluate it because using samples to answer a query will bring a loss on the query accuracy. The relative error is calculated by Formula (1)

$$RE(q|\widetilde{S}) = \frac{|\theta(D) - \hat{\theta}(\widetilde{S})|}{\theta(D)} \tag{1}$$

, where $\theta(D)$ is the exact query result based on the underlying data D and $\hat{\theta}(\widetilde{S})$ is the approximate query result based on the active sample $\widetilde{S}$. For group-by queries, the relative error is the average error upon each group. Therefore, the benefit of using an active sample $\widetilde{S}$ to answer query q is defined as follows.

Definition II.1 (Benefit)**.** *Given a query q and an active sample $\widetilde{S}$, the benefit of using $\widetilde{S}$ to answer q is defined as*

$$B(q|\widetilde{S}) = \begin{cases} \alpha \cdot RL(q|\widetilde{S}) + (1-\alpha) \cdot -RE(q|\widetilde{S}), & if\ \widetilde{S} \rightarrow q \\ -1, & otherwise \end{cases} \tag{2}$$

, where the first part is the improvement on query latency, the second part is the improvement on query accuracy, and α is set by users according to their requirements for the query accuracy. If $\widetilde{S}$ cannot support q, the benefit is set to be -1.

Cost of an active sample. The cost of an active sample $\widetilde{S}$ is $C(\widetilde{S}) = C_g(\widetilde{S}) + C_l(\widetilde{S})$, where $C_g(\widetilde{S})$ is the time cost of generating $\widetilde{S}$ from the underlying data and $C_l(\widetilde{S})$ is the time cost of loading $\widetilde{S}$ from the sample pool to the sample buffer. Note that we consider $C_g(\widetilde{S})$ in the calculation of $C(\widetilde{S})$ because the more chance the active samples can be selected to reuse, the more the cost of sample generation can be amortized.

Benefit of an active sample set. Suppose we have an active sample set $\widetilde{\mathbb{S}}$. To evaluate the benefit brought by this set of active samples for a specific query q, we define $B(q|\widetilde{\mathbb{S}})$. Note that we use only one active sample to answer the query, even if there are multiple active samples that can be used to answer q. This strategy is also widely used in existing AQP systems [7], [16]. Hence, we define $B(q|\widetilde{\mathbb{S}})$ as the maximum benefit brought by samples in $\widetilde{\mathbb{S}}$, i.e., $B(q|\widetilde{\mathbb{S}}) = \max_{\widetilde{S}\in\widetilde{\mathbb{S}}} B(q|\widetilde{S})$.

Utility of an active sample. Suppose we have an active sample $\widetilde{S} \in \widetilde{\mathbb{S}}$. For a workload $\mathbb{Q}$, the queries that can be answered by $\widetilde{S}$ to get the maximum benefit is denoted as $\mathbb{Q}_{\widetilde{S}} = \{q|q \in \mathbb{Q} \wedge \widetilde{S} = \widetilde{S}_q\}$, where $\widetilde{S}_q$ is the active sample in $\widetilde{\mathbb{S}}$ that can bring the maximum benefit for q. If none of the active samples in $\widetilde{\mathbb{S}}$ can be used to answer the query q, we will set $\widetilde{S}_q$ as null. The utility of the active sample $\widetilde{S}$ is calculated by the total benefit of using $\widetilde{S}$ for $\mathbb{Q}$ minus the cost of $\widetilde{S}$. To use the same metric as the definition of benefit, we use the relative cost instead of the actual cost of $\widetilde{S}$ in the following definition.

Definition II.2 (Utility)**.** *Given a workload $\mathbb{Q}$ and an active sample $\widetilde{S}$, the utility of using an active sample $\widetilde{S}$ is defined as follows:*

$$U(\widetilde{S}|\mathbb{Q}) = \sum_{q\in\mathbb{Q}_{\widetilde{S}}} B(q|\widetilde{S}) - \frac{C(\widetilde{S})}{\sum_{q\in\mathbb{Q}_{\widetilde{S}}} (L(q|D) - L(q|\widetilde{S})) + 1} \quad (3)$$

, where the first part is the total benefit of using $\widetilde{S}$ and the second part is the relative cost between the cost of $\widetilde{S}$ and the query latency improvement of using $\widetilde{S}$. To avoid the zero division error when $\mathbb{Q}_{\widetilde{S}}$ is empty, we add 1 to the denominator of the second part.

Utility of an active sample set. Suppose we have an active sample set $\widetilde{\mathbb{S}}$. For each query q in the workload $\mathbb{Q}$, we select an active sample $\widetilde{S}_q \in \widetilde{\mathbb{S}}$ that can bring the maximum benefit for q. The $\widetilde{S}_q$ is set as null if none of the active samples in $\widetilde{\mathbb{S}}$ can be used to answer the query q. To evaluate the utility of using $\widetilde{\mathbb{S}}$ for the workload $\mathbb{Q}$, we define $U(\widetilde{\mathbb{S}}|\mathbb{Q})$, which is calculated by Formula (4).

$$U(\widetilde{\mathbb{S}}|\mathbb{Q}) = \sum_{q\in\mathbb{Q}} B(q|\widetilde{S}_q) - \frac{\sum_{\widetilde{S}\in\widetilde{\mathbb{S}}} C(\widetilde{S})}{\sum_{q\in\mathbb{Q}} (L(q|D) - L(q|\widetilde{S}_q)) + 1} \quad (4)$$

In Formula (4), the first part is the total benefit obtained from all queries by using $\widetilde{\mathbb{S}}$ and the second part is the relative cost between the total cost of $\widetilde{\mathbb{S}}$ and the query latency improvement achieved by all queries in the workload $\mathbb{Q}$. Note that we ignore the queries whose $\widetilde{S}_q$ is null when calculating the utility of an active sample set by Formula (4).

Sample tuning. Given a query q, sample tuning will be triggered if none of the active samples in the sample buffer can be used to answer q or the accuracy of the approximate query result cannot meet the user's requirement about query accuracy. When tuning the sample buffer, we aim to select an optimal active sample set to maximize the utility for the current workload $\mathbb{Q}$, meanwhile keeping the storage size of the active sample set below the given storage budget. In this paper, we define this problem as an *Active Sample Selection* problem, which can be reduced to the NP-hard knapsack constraint problem because each sample can be used for answering more than one query, and some queries can be answered by more than one sample [13].

Definition II.3 (Active Sample Selection problem)**.** *Suppose we have a sample set $\mathbb{S}$. Given a storage budget M_{max} of the sample buffer, the problem is to select an optimal active sample set $\widetilde{\mathbb{S}}$ that has the maximum utility according to the given workload $\mathbb{Q}$. The problem can be formulated as follows:*

$$\begin{aligned} &\arg\max_{\widetilde{\mathbb{S}}\subseteq\mathbb{S}} U(\widetilde{\mathbb{S}}|\mathbb{Q}) \\ s.t. \quad &M_{\widetilde{\mathbb{S}}} = \sum_{\widetilde{S}\in\widetilde{\mathbb{S}}} m_{\widetilde{S}} \le M_{max} \end{aligned} \quad (5)$$

, where $m_{\widetilde{S}}$ is the storage size of an active sample $\widetilde{S}$ and $M_{\widetilde{\mathbb{S}}}$ is the total storage size of $\widetilde{\mathbb{S}}$. Specifically, the sample set $\mathbb{S}$ is generated based on $\mathbb{Q}$ by the sample generator.

III. DEEP REINFORCEMENT LEARNING-BASED SAMPLE TUNER

A. DQN-based Sample Selection Approach

To avoid the heavy overhead of selecting the actual optimal active sample set for the given workload, *RL-STuner* uses a Deep Q-learning Network (DQN) model [15] to solve the *Active Sample Selection* problem. When selecting samples, we use a sample state z_S to indicate whether a sample S in the sample set $\mathbb{S}$ is selected into the sample buffer. If S is selected into the sample buffer, let $z_S = 1$. Otherwise, let $z_S = 0$. We can easily get the selected samples that will be stored in the sample buffer in terms of the sample state z_S of each sample. During the sample selection, we regard a selected sample S with $z_S = 1$ as an active sample $\widetilde{S} = S|S \in \mathbb{S} \wedge z_S = 1$ and a set of selected samples as an active sample set, i.e., $\widetilde{\mathbb{S}} = \{\widetilde{S}\} = \{S|S \in \mathbb{S} \wedge z_S = 1\}$.

The process of solving the *Active Sample Selection* problem can be transformed as a Markov Decision Process (MDP), since the new sample state set only depends on the current sample state set and the samples whose state will be flipped. As shown in Figure 3, we map the process of active sample selection into MDP as follows.

- **Environment.** Environment is the AQP system with the workload. In the environment, the AQP system uses the active samples in the sample buffer to answer all the queries in the workload.

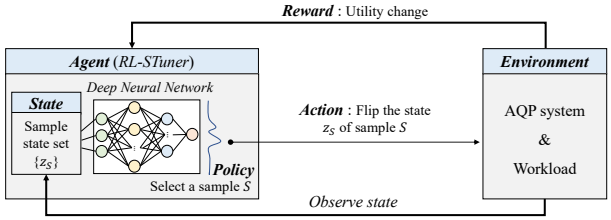


Fig. 3: The Markov Decision Process framework of the Active Sample Selection problem

- **Agent.** Agent is the *RL-STuner*. By receiving rewards and states from the environment, *RL-STuner* updates the policy to guide the sample selection for an optimal sample set.
- **State.** State e is a set of sample state that represents whether a sample $S \in \mathbb{S}$ is selected to store in the sample buffer, i.e., $e = \{z_S\}$. In addition, we denote the utility of using active samples at state e as $U(e) = U(\widetilde{\mathbb{S}}|\mathbb{Q})$, where $\widetilde{\mathbb{S}}$ is the set of samples with sample state $z_S = 1$ in e.
- **Action.** Action a is to flip the state of a given sample S, i.e., flip z_S from $0/1$ to $1/0$.
- **Policy.** Policy defines which sample will be flipped in a specific state and environment. In other words, given a state e, the policy will select a sample S from $\mathbb{S}$.
- **Reward.** Reward is the utility change received after transitioning from the state e to the new state e' due to the action a, calculated by the formula $R_a(e, e') = U(e') - U(e)$.

Therefore, the *Active Sample Selection* problem becomes a reinforcement learning problem, whose goal is to learn an optimal policy to select the active samples, which is defined as follows:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi}[\sum_{t=0}^{T-1} \gamma^t R_{a_t}(e_t, e_{t+1})] \tag{6}$$

, where π represents a policy function that selects an action according to a given state, $\mathbb{E}_{\pi}[\cdot]$ represents the expected total discounted reward of the policy π for a given state, $\gamma \in [0, 1)$ represents the discount rate, t is the time step, e_t is the state at time t, and a_t is the action selected by policy π at time t.

Since both the state space and the action space in the *Active Sample Selection* problem have an exponential relationship with the number of samples in the sample pool, we use the DQN model to guide the sample selection in *RL-STuner*. DQN is a value-based reinforcement learning algorithm that trains a deep neural network to approximate the state-action value Q-value of each state and action. For a given state e and action a, Q-value is the cumulative reward of performing action a at state e, which is denoted as $Q(e, a)$. The higher the $Q(e, a)$, the better the action a is for the state e. Hence, we can get the best policy to select samples based on the Q-values. By using the neural network, DQN can deal efficiently with the curse of dimensionality, unlike Q-learning [17].

Figure 4 shows the overview of the DQN-based sample selection approach (abrr. DQN-SS) in *RL-STuner*. The in-

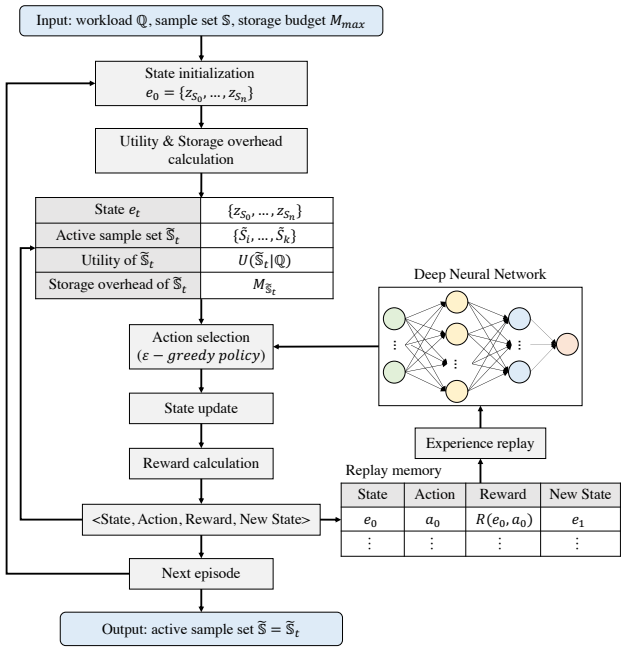


Fig. 4: An overview of the DQN-based sample selection approach in *RL-STuner*

put of the DQN-based sample selection approach includes a workload $\mathbb{Q}$, a sample set $\mathbb{S}$ that is generated based on the queries in $\mathbb{Q}$ as described in Section II-A, and a storage budget M_{max} of sample buffer. With these inputs, we first initialize a state $e_0 = \{z_{S_0}, ..., z_{S_n}\}$. According to the state e_0, we get the current active sample set $\widetilde{\mathbb{S}}_0$ and calculate the utility and storage overhead of $\widetilde{\mathbb{S}}_0$. Then, we use the ϵ-greedy policy to select an action a_0 according to the state $e_t = e_0$. The ϵ-greedy policy is an action selection method to balance exploration and exploitation. If the value of ϵ is less than a random value, we randomly select an action from the action space, i.e., randomly select a sample S from the sample set $\mathbb{S}$ to flip its state. Otherwise, we select the action that has the maximum Q-value for state e_0 by using the deep neural network, i.e., $a_0 = \arg\max_a Q(e_0, a)$. Next, according to the selected action, we update the state $e_t = e_1$, the active sample set $\widetilde{\mathbb{S}}_t = \widetilde{\mathbb{S}}_1$, the utility of the active sample set, and the storage overhead of the active sample set. We also calculate the reward between the state e_0 and the new state e_1. After that, we store the state, the selected action, the reward, and the new state in the replay memory and use these experiences to update the deep neural network. These processes will be repeated in an episode until the time step of this episode is larger than the given threshold or the storage size of the active sample set is larger than the given storage budget. Finally, when the maximum allowed number of episodes is reached, we return the last active sample set $\widetilde{\mathbb{S}}_t$ as the selected active sample set to store in the sample buffer. The time complexity of DQN-SS is about $O(E \cdot T \cdot |\mathbb{Q}| \cdot |\widetilde{\mathbb{S}}|)$, where E is the number of episodes in DQN, T is the number of time steps in DQN, and $|\mathbb{Q}| \cdot |\widetilde{\mathbb{S}}|$ is the time complexity of calculating the utility $U(\widetilde{\mathbb{S}}|\mathbb{Q})$. Note that we ignore the time complexity of the deep neural network in DQN-SS because of its concise structure that has four fully

connected layers.

Since the DQN-based sample selection approach in *RL-STuner* has the memory ability to store the sample selection experiences in different episodes, it can use these experiences to guide sample selection to approximate the global optimal result. By sharing the sample selection experiences in episodes, *RL-STuner* knows which action is a good choice for the given state even if it has not performed this action in the current episode. Furthermore, *RL-STunecr* can also utilize the prior knowledge of workload to improve the AQP performance. According to the given prior knowledge of the query workload, *RL-STuner* can select a set of active samples into the sample buffer in advance offline and reuse them across future queries.

B. Optimization Mechanisms for Tuning Cost

1) Lazy Sample Tuning: To adapt to the changing workload in IDE, we will tune the sample buffer if none of the active samples in the sample buffer can be used to answer the new query q or the accuracy of the approximate query result cannot meet the user's requirement about query accuracy. The natural sample tuning strategy is full sample tuning strategy, which re-selects a new active sample set $\widetilde{\mathbb{S}}'$ to replace all active samples in the sample buffer when the workload changes. However, the cost of the full sample tuning strategy is expensive because the calculation cost of sample selection increases as the workload size increases and the I/O cost of replacing all active samples in the sample buffer is large. Fortunately, we find that some sample selection and tuning operations in the full sample tuning strategy are unnecessary since there is often an overlap between the current active sample set $\widetilde{\mathbb{S}}$ and the new active sample set $\widetilde{\mathbb{S}}'$. The difference between $\widetilde{\mathbb{S}}$ and $\widetilde{\mathbb{S}}'$ is that some active samples in $\widetilde{\mathbb{S}}'$ is used to answer the queries which trigger the sample tuning for the current active sample set $\widetilde{\mathbb{S}}$. Therefore, to reduce the sample tuning cost, we propose a lazy sample tuning strategy (abbr. LAZY) to avoid unnecessary sample selection and tuning operations.

The lazy sample tuning strategy works as Algorithm 1 shown. The input of this algorithm includes a workload $\mathbb{Q}$, a sample set $\mathbb{S}$, an active sample set $\widetilde{\mathbb{S}}$, a storage budget M_{max} of sample buffer, and a relative error threshold β that represents the user's requirement about query accuracy. With these inputs, we first find out which active samples will be saved in the sample buffer (line 2-5). For each active sample $\widetilde{S}$ in $\widetilde{\mathbb{S}}$, we remove it from $\widetilde{\mathbb{S}}$ if it has the minimum utility or it can only be used to answer one query in $\mathbb{Q}$. The removed active sample set and saved active sample set are denoted as $\widetilde{\mathbb{S}}_r$ and $\widetilde{\mathbb{S}}_s$, respectively. Then, we find out the queries $\mathbb{Q}_t$ which will trigger the sample tuning for $\widetilde{\mathbb{S}}_s$ (line 6-8). Next, by using the DQN-based sample selection approach in Section III-A, we select an optimal active sample set $\widetilde{\mathbb{S}}_t$ for $\mathbb{Q}_t$ (line 9-11). Finally, we store $\widetilde{\mathbb{S}}_t$ in sample buffer and get the tuned active sample set $\widetilde{\mathbb{S}}' = \widetilde{\mathbb{S}}_s \cup \widetilde{\mathbb{S}}_t$ (line 12-13). The time complexity of the lazy sample tuning strategy is about $O(|\widetilde{\mathbb{S}}|+|\mathbb{Q}|\cdot|\widetilde{\mathbb{S}}_s|+E\cdot T\cdot|\mathbb{Q}_t|\cdot|\widetilde{\mathbb{S}}_t|)$. Obviously, the sample tuning cost of the lazy strategy is much lower than that of the full sample tuning strategy because $|\mathbb{Q}_t|$ is much less than $|\mathbb{Q}|$ and we only need to select a few active samples $\widetilde{\mathbb{S}}_t$ when tuning

Algorithm 1: Lazy sample tuning (LAZY)

Input: workload $\mathbb{Q}$, sample set $\mathbb{S}$, active sample set $\widetilde{\mathbb{S}}$, storage budget M_{max} of sample buffer, relative error threshold β.
Output: tuned active sample set $\widetilde{\mathbb{S}}'$.

1 $\widetilde{\mathbb{S}}_r = \emptyset, \mathbb{Q}_t = \emptyset$;
2 **foreach** $\widetilde{S} \in \widetilde{\mathbb{S}}$ **do**
3 **if** $\widetilde{S} == \arg\min_{\widetilde{S}\in\widetilde{\mathbb{S}}} U(\widetilde{S}|\mathbb{Q})$ *or* $|\mathbb{Q}_{\widetilde{S}}| == 1$ **then**
4 $\widetilde{\mathbb{S}}_r.add(\widetilde{S})$;
5 $\widetilde{\mathbb{S}}_s = \widetilde{\mathbb{S}} - \widetilde{\mathbb{S}}_r$;
6 **foreach** $q \in \mathbb{Q}$ **do**
7 **if** $\min_{\widetilde{S}\in\widetilde{\mathbb{S}}_s} RE(q|\widetilde{S}) > \beta$ **then**
8 $\mathbb{Q}_t.add(q)$;
9 $\mathbb{S}_c = \mathbb{S} - \widetilde{\mathbb{S}}_s$;
10 $M_f = M_{max} - M_{\widetilde{\mathbb{S}}_s}$;
11 $\widetilde{\mathbb{S}}_t =$ call the DQN-based sample selection approach to select an optimal active sample set with $\mathbb{Q}_t, \mathbb{S}_c, M_f$;
12 $\widetilde{\mathbb{S}}' = \widetilde{\mathbb{S}}_s \cup \widetilde{\mathbb{S}}_t$;
13 return $\widetilde{\mathbb{S}}'$;

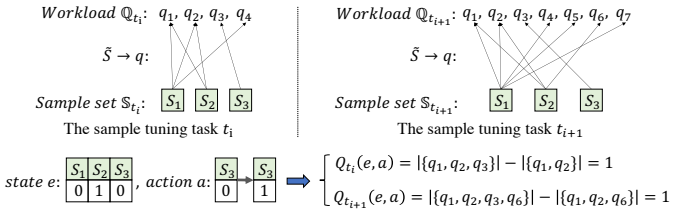


Fig. 5: An example of the same Q-value in two consecutive sample tuning tasks

samples. The lazy strategy needs additional space to store the utility of each active sample and the number of times the active sample was used. However, the space cost can be ignored because the number of active samples in the sample buffer is limited. Since both $\widetilde{\mathbb{S}}_s$ and $\widetilde{\mathbb{S}}_t$ have a good performance for the corresponding queries in $\mathbb{Q}$, $\widetilde{\mathbb{S}}'$ also has a good performance for $\mathbb{Q}$ even if it is not the optimal active sample set for $\mathbb{Q}$. In fact, the lazy sample tuning strategy makes a trade-off between the query accuracy and the sample tuning cost.

2) Parameter Transfer: To obtain a strategy with high utilities, we often need to use a large number of episodes to train the DQN model. However, the larger number of episodes, the higher cost of the model training. Fortunately, when tuning samples, we find that some state-action pairs in the DQN model have the similar Q-values across two consecutive sample tuning tasks, because the workloads between these tasks are very similar in *RL-STuner*. Figure 5 shows an example of the same Q-value in two consecutive sample tuning tasks. Given two consecutive sample tuning tasks t_i and t_{i+1}, the workloads in these tasks are $\mathbb{Q}_{t_i} = \{q_1, q_2, q_3, q_4\}$ and $\mathbb{Q}_{t_{i+1}} = \{q_1, q_2, ..., q_7\}$, and the sample sets in these tasks are $\mathbb{S}_{t_i} = \mathbb{S}_{t_{i+1}} = \{S_1, S_2, S_3\}$. For simplicity, we set

the utility of using an active sample set $\widetilde{\mathbb{S}}$ for a workload $\mathbb{Q}$ as $U(\widetilde{\mathbb{S}}|\mathbb{Q}) = |\{q_i|q_i \in \mathbb{Q} \wedge \widetilde{S}_{q_i} \neq null\}|$. As shown in Figure 5, for the state-action pair (e, a) which has appeared in both t_i and t_{i+1}, since the sample S_3 cannot be used to answer any queries in $\mathbb{Q}_{t_i} - \mathbb{Q}_{t_{i+1}} = \{q_5, q_6, q_7\}$, the state-action value $Q(e, a)$ is same across the two sample tuning tasks, i.e., $Q_{t_i}(e, a) = Q_{t_{i+1}}(e, a)$. Therefore, inspired by the parameter transfer in transfer learning [18], we propose a parameter transfer (abbr. TRAN) approach to initialize the DQN model for the subsequent sample tuning task by the DQN model for the previous sample tuning task to speed-up the learning process.

However, since the state-action space in the DQN-based sample selection approach is set based on the sample set in the sample pool when sample tuning, the DQN models may have different state-action spaces in two sample tuning tasks. Hence, the knowledge in the historical sample tuning task cannot be directly used by the DQN model to learn on the new sample tuning task. To solve this problem, we define a mapping between the state-action spaces of DQN models in the different sample tuning tasks to transfer knowledge effectively. For each sample tuning task, we set the state-action space in the DQN model according to a given parameter rather than the number of samples in the sample pool. Therefore, the DQN models in the different sample tuning tasks have the same state-action space even if they have different inputs. Moreover, to avoid generating a wrong action on the non-existent samples, we set the search space of the ϵ-greedy algorithm in each sample tuning task according to the sample set in the sample pool. The parameter transfer approach also supports sample updates. For sample updates, we only need to recalculate the Q-values in the state-action pairs which are related to the updated samples.

3) Utility Estimation: To calculate the utility of using an active sample set $\widetilde{\mathbb{S}}$ for the workload $\mathbb{Q}$, we need to get the exact result and latency of each query q in $\mathbb{Q}$ when tuning samples. However, the time cost of executing queries on the underlying dataset is expensive. Therefore, to reduce the time cost of calculating sample utility, we propose a *utility estimator* (abbr. EST), which calculates an approximate value of the utility without executing queries on the underlying dataset.

In the utility estimator, we use the confidence interval of the approximate query result to get an approximate value of the relative error. For an active sample $\widetilde{S}$ and a query q that can be answered by $\widetilde{S}$, the approximate relative error is calculated as $\hat{RE}(q|\widetilde{S}) = |CI_{up}(\widetilde{S}) - \hat{\theta}(\widetilde{S})|/\hat{\theta}(\widetilde{S})$, where $CI_{up}(\widetilde{S})$ is the upper bound of the confidence interval. Since the confidence interval of the approximate query result usually covers the exact query result, the approximate relative error usually is an upper bound of the true relative error. Specifically, we use the standard closed-form formulas [19] to estimate the confidence interval for aggregate functions because the time cost of the closed-form estimate method is much lower than that of the bootstrap method [20]. For the query latency on the underlying dataset, the utility estimator uses the optimizer in the database management system to estimate the query latency. In this paper, we use the EXPLAIN command in PostgreSQL.

C. Support for Data Updates

For a data warehouse, data updates are usually data appending operations. Hence, in this paper, we only consider the data update form of data appending. For data updates, *RL-STuner* does not need to modify its tuning functionality, but need to update the samples in the sample buffer and sample pool. When new data arrives, we have three sample update strategies: 1) *Timely*, which will update all samples timely in both sample buffer and sample pool; 2) *Delayed*, which will update the samples in the sample buffer timely and delay the sample updates on the sample pool, until the size of new data exceeds a given threshold; 3) *Batch*, which will update all samples in batch when the size of new data exceeds a given threshold. Although the timely sample update strategy can keep the samples fresh so that it can provide query accuracy guarantee to some extent, it will cause a heavy overhead for each sample update. For the batch update strategy, it reduces the number of sample updates while it causes a loss of query accuracy due to the stale samples. The delayed update strategy tries to find a balance between the timely and the batch strategies. Hence, in this paper, we take the delayed sample update strategy to support data updates. **When updating samples, we can use the R* algorithm [21] if the sample size is fixed.** If the user gives a requirement about query accuracy, we will use the timely update strategy and the adaptive sample update approach [22] to update samples to provide a priori query accuracy guarantee for the user as much as possible. Furthermore, the three update strategies can also be used to update samples when deleting or updating tuples in the underlying dataset. **For the data deleting operation, since the R* algorithm just marks the deleted tuples in a sample as invalid rather than removing these tuples from this sample, we need to update the sampling ratio of this updated sample accordingly to guarantee query accuracy.** For the data updating operation, we implement it through a combination of the deleting operation and the appending operation.

IV. EXPERIMENTS

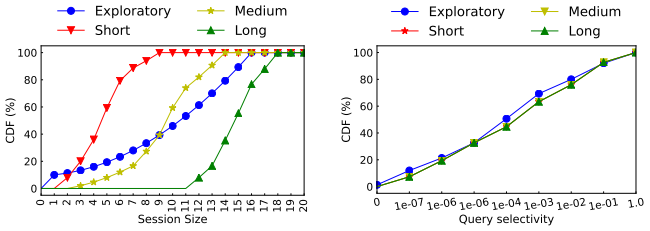
A. Experiment Setup

Experimental Settings. We implement an AQP system on top of PostgreSQL 11.2 to support AQP queries. The AQP system is responsible for query rewriting, answer rewriting, and sample tuning. PostgreSQL is used to get the query results on samples. For the active samples in the sample buffer, we store them in the memory by using the "pg_prewarm" module[2]. The underlying data and the samples in the sample pool are stored in the disk. All experiments are conducted on a Linux machine with Intel Xeon Gold 5215 CPU, Nvidia Titan RTX GPU, 64GB RAM, and 3.3TB HDD disk. Note that the GPU is only used to train the DQN model in RL-STuner.

Approaches. We implement four sample tuning approaches in the AQP system and compare them by the experiments.

- *Taster*: a sample tuning approach that uses a sliding window of the previous w queries as an approximation of

[2]https://www.postgresql.org/docs/current/pgprewarm.html



(a) Session size distribution (b) Query selectivity distribution

Fig. 6: Workload patterns on Flighs dataset

future queries to guide sample tuning [13]. When tuning samples, Taster uses a greedy algorithm [14] to calculate the sample set which has the maximum utility for the w recent queries. The default value of w is set at 10.

- *Taster-AW*: an extension of *Taster* that uses an adaptive window (abbr. AW) instead of a fixed-size window by dynamically changing w according to the workload [13].
- *Taster-FW*: an extension of *Taster* that uses a full-size window (abbr. FW) instead of a fixed-size window by considering all the queries that have appeared in the workload, i.e., w is set at the number of queries in the workload. Therefore, when tuning samples, Taster-FW can leverage all the queries in the workload that have been input into the AQP system to guide the sample tuning.
- *RL-STuner*: a sample tuning approach that integrates a DQN-based sample selection approach with optimization mechanisms proposed in this paper. The number of episodes in the DQN model is set at 500.

Datasets. We conduct experiments on two datasets. 1) *Flights (real dataset)*: a real-world dataset [23] that has 12 attributes and contains information about on-arrival statistics for the last few years. By default, we use the data generator from [2] to scale the dataset up to 100 million tuples while ensuring that the relationships between attributes are still maintained. 2) *TPC-H (synthetic dataset)*: to demonstrate the effectiveness of our approach for complex queries with join clauses and group-by clauses. We conduct experiments on a TPC-H dataset [24] with a fact table and 7 dimension tables. By default, the fact table has 100 million tuples.

Workloads. We use the state-of-the-art IDE benchmark, IDEBench [2], to generate workloads on the above two datasets for interactive data exploration. Each workload includes 150 queries.

1) Workload on Flights: we use IDEBench [2] to generate an *exploratory* workload, which is composed of 15 different size of sessions and 15 ad-hoc queries. The queries in the workload have different predicates on different attributes and the session size distribution is shown in Figure 6(a)[3]. Furthermore, to evaluate the effect of session size on the performance, we generate three workloads with different session sizes: *short session* workload, *medium session* workload, and *long session* workload. To avoid the interference from ad-hoc queries, we exclude the ad-hoc queries from these workloads. Figure 6(a) shows the session size distribution of these three workloads.

[3]Let an ad-hoc query equivalent to a session whose size is 1.

Note that to compare the performance fairly, we keep the queries in these three workloads staying the same, while distributing these queries into different sessions. As shown in Figure 6(b), the queries in the workload covers various selectivities that is closer to the real application scenario. By default, our experiments are conducted on the *exploratory* workload.

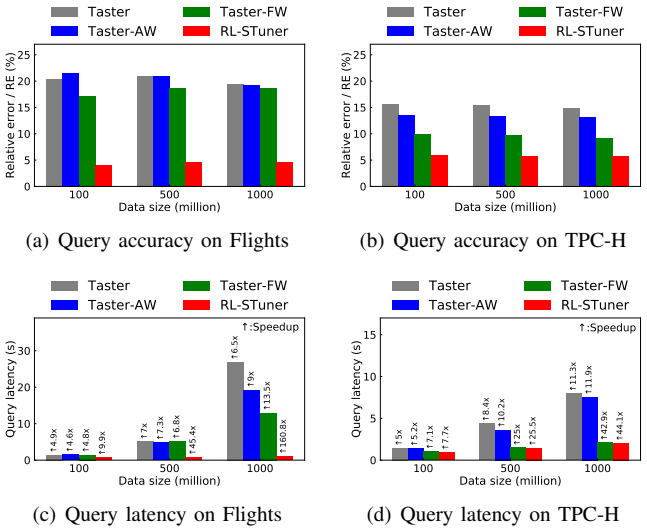
2) Workload on TPC-H: we use IDEBench [2] to generate an *exploratory* workload for TPC-H, which is also composed of 15 different size of sessions and 15 ad-hoc queries. Since IDEBench [2] cannot generate complex queries with join condition, in order to evaluate the performance of complex queries, we use the join clauses in the typical complex queries Q_{12} and Q_{14} from the TPC-H benchmark to extend twelve queries in the *exploratory* workload.

Performance Metrics. We use two metrics to evaluate the performance. 1) *Relative Error (RE)*: which is calculated by Formula (1) to evaluate the query accuracy. 2) *Query Latency*: which is the query response time. Futhermore, we also evaluate the performance improvement of these methods by using the *Speedup* metric that is calculated by $Speedup = L_{EXACT}/L_{AQP}$, where L_{EXACT} is the query latency of getting the exact query result on the underlying dataset by using PostgreSQL and L_{AQP} is the query latency of getting the approximate query result by using AQP. Note that for the query that cannot be answered by the active samples in the sample buffer, we set $L_{AQP} = L_{EXACT}$.

In the following experiments, the relative error, the query latency, and the speedup is the average value of 10 separate experiments with the same experimental settings. By default, the sampling ratio is set at 10%, the storage budget of the sample buffer is set at 50% of the underlying data size, the relative error threshold above which sample tuning will be triggered is set at 2%, the tuning time budget of the sample tuning approaches is set at 60 seconds, and no prior knowledge of the workloads is provided.

B. Comparison of Sample Tuning Approaches

Performance on Flights and TPC-H datasets. We compare the performance of Taster, Taster-AW, Taster-FW, and RL-STuner on both a real dataset (Flights) and a synthetic dataset (TPC-H) with different data sizes. We vary the data size from 100 million tuples to 1 billion tuples and set the tuning time budget according to the data size. Figure 7(a) and Figure 7(b) show the average relative error of 150 queries in the exploratory workload by using four different sample tuning approaches. We find that the data size has a little effect on the query accuracy of the four approaches, and RL-STuner outperforms other approaches. For Flights, RL-STuner achieves about 4×-5.2 ×improvements over Taster, Taster-AW, and Taster-FW on the query accuracy. For TPC-H, RL-STuner achieves about 1.6×-2.7× improvements over Taster, Taster-AW, and Taster-FW on the query accuracy. RL-STuner has a higher query accuracy than Taster and Taster-AW because its tuning strategy from a global perspective makes it can select appropriate samples to answer the queries with high accuracy. Furthermore, RL-STuner outperforms Taster-FW even though



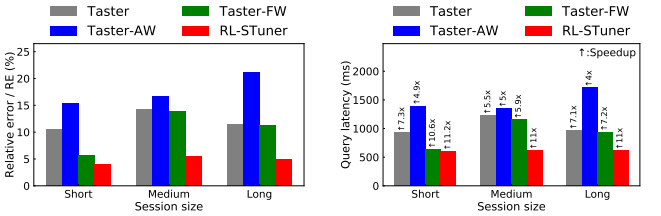
(a) Query accuracy on Flights (b) Query accuracy on TPC-H

(c) Query latency on Flights (d) Query latency on TPC-H

Fig. 7: Performance on Flights and TPC-H datasets

TABLE I: The average time cost of sample tuning on Flights and TPC-H datasets

	Data Size	Taster	Taster-AW	Taster-FW	RL-STuner
Flights	100 million	48s	52s	55s	21s
	500 million	158s	127s	164s	47s
	1 billion	295s	316s	360s	78s
TPC-H	100 million	50s	55s	68s	20s
	500 million	156s	160s	171s	51s
	1 billion	340s	341s	363s	82s



(a) Query accuracy (b) Query latency

Fig. 8: Performance for workloads with various session sizes on Flights dataset

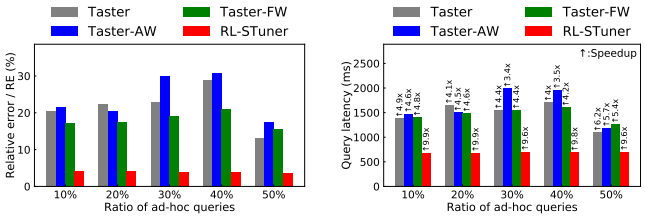
Taster-FW also tunes samples from the global perspective. This is because RL-STuner can get a global optimal solution by using the DQN model, while Taster-FW just performs a local optimization by using the greedy algorithm. Note that the local optimization in Taster-FW means it cannot get a global optimal solution for the given problem.

Figure 7(c) and Figure 7(d) show the average query latency and speedup of 150 queries in the exploratory workload by using four different sample tuning approaches. Overall, RL-STuner has a lower query latency and a higher speedup than Taster, Taster-AW, and Taster-FW since the active samples tuned by RL-STuner can be used to answer new queries when the workload changes. The average query latency of Taster, Taster-AW, and Taster-FW increases significantly as the data size increases. This is because the AQP system needs more time cost to access the underlying dataset to get the results for the queries that cannot be answered by the active samples. The query latency of RL-STuner is also affected by the data size, but the effect is much less because there are more chances that the active samples in the memory selected by RL-STuner can cover more queries. The query latency of Taster-FW on TPC-H is less than that on Flights because the data distribution in Flights is very skewed, which makes the active samples tuned by Taster-FW unable to adapt to the changes in workload well. As for the sample tuning cost, as the data size grows, so does the sample tuning cost, as shown in Table I. However, the sample tuning cost of RL-STuner is still much lower than that of Taster, Taster-AW, and Taster-FW. The results shown in the above experiments demonstrate that RL-STuner can not only achieve better performance on query accuracy and latency when facing workload changes but also has a lower tuning cost than other sample tuning approaches.

Moreover, we evaluate the gap between RL-STuner and the theoretical optimal solution. Since the time cost of getting the optimal solution is expensive, we limit the maximum size of the sample pool to 40 samples and compare the performance of the optimal solution and RL-STuner on Flights (100 million tuples). The sample tuning cost of getting the optimal solution is too high (about 2 hours) to apply to the real AQP system. **In contrast, the sample tuning cost of RL-STuner is only about 20 seconds, while it only degrades the utility of active samples by about 13.6% of the optimal solution and the query accuracy by about 10% of the optimal solution.**

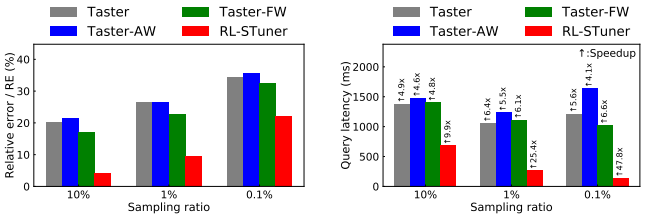
Effect of workload changes. To evaluate the effect of workload changes, we conduct experiments by using multiple workloads with different characteristics on the Flights dataset. We first use three workloads (the short session workload, the medium session workload, and the long session workload) that have different session sizes to evaluate the effect of session size on the performance. The detailed session size distribution of the three workloads is shown in Figure 6(a).

Figure 8(a) shows the average relative error of the queries in the three workloads. Overall, RL-STuner has a lower relative error than Taster, Taster-AW, and Taster-FW. The session size has a limited effect on the relative error of RL-STuner since it tunes samples from a global perspective and obtains the global optimal solution by using the DQN model, which keeps it from the predicament of optimization for a specific session. On the contrary, the session size has a large effect on the relative error of Taster, Taster-AW, and Taster-FW. For Taster and Taster-AW, the relative error of them on the medium session workload (14.2%/16.7%) and long session workload (11.5%/21.2%) is larger than that on the short session workload (10.5%/15.3%), because they are easily stuck in the optimization for a specific session when the session size is close to the window size w. Specifically, the relative error of Taster on the long session workload is less than that on the medium session workload because the assumption in Taster is easier to hold in the long session workload. The relative error of Taster-AW on the long session workload does not benefit from this because the adaptive window size is tuned too small in the process of



(a) Query accuracy (b) Query latency

Fig. 9: Performance for workloads with various ratios of ad-hoc queries on Flights dataset



(a) Query accuracy (b) Query latency

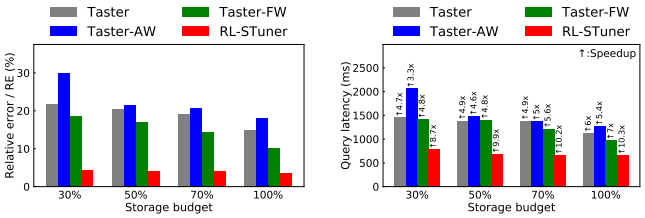
Fig. 10: Performance with different sampling ratios on Flights dataset

auto-tuning. The increased session size also decays the query accuracy of Taster-FW even though it tunes samples from a global perspective, since it cannot converge into the global optimal solution by using the greedy algorithm. As shown in Figure 8(b), the same reason causes the session size to have a limited effect on the query latency of RL-STuner, but Taster, Taster-AW, and Taster-FW cannot ignore the effect of the session size.

During the data exploration, except for sessional queries, users often issue ad hoc queries aperiodically. Hence, to evaluate the effect of ad-hoc queries on performance, we vary the ratio of ad-hoc queries in the exploratory workload from 10% to 50%. Specifically, to compare the performance fairly, we shuffle some sessional queries in the exploratory workload as ad-hoc queries. Thus, in this experiment, although the ratios of ad-hoc queries in these workloads are different, they have the same set of queries. Figure 9(a) shows the average relative error of the queries by using the four different sample tuning approaches. Overall, RL-STuner has a lower relative error than Taster, Taster-AW, and Taster-FW. As shown in Figure 9(a), the increased ad-hoc queries will not increase the relative error of RL-STuner. The ratio of ad-hoc queries has a limited effect on the relative error of RL-STuner since it uses all queries in the current appeared workload to tune samples, which makes it can obtain the relationship between the queries even though they are far apart. On the contrary, the ratio of ad-hoc queries has a larger effect on the relative error of Taster, Taster-AW, and Taster-FW. For Taster and Taster-AW, this is because the ratio of ad-hoc queries will decide whether the new queries depend on the recent queries. Specifically, Taster and Taster-AW have a lower relative error on the 50% ratio of ad-hoc queries because these ad-hoc queries decrease the number of sessional queries and the session size. Hence, these two approaches are easily getting out of the predicament that is only optimized for a long session. For Taster-FW, this is because it uses the same greedy algorithm as Taster and Taster-AW to select the active samples when tuning samples. The same reason causes the ratio of ad-hoc queries to have a limited effect on the query latency of RL-STuner, but Taster, Taster-AW, and Taster-FW cannot ignore the effect of the ratio of ad-hoc queries, as shown in Figure 9(b).

In summary, compared with Taster, Taster-AW, and Taster-FW, RL-STuner can better adapt to the workload changes in session size and the ratio of ad-hoc queries.

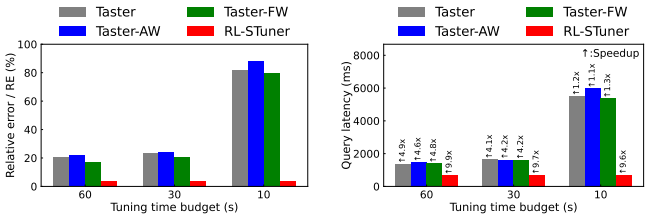
Effect of sampling ratio. To evaluate the effect of sampling ratio on the performance, we vary the sampling ratio from 10% to 0.1% and conduct experiments for the exploratory workload on the Flights dataset. Note that the size of the sample buffer is fixed in these experiments. As shown in Figure 10(a), the average relative error of queries by using the four sample tuning approaches increases as the sampling ratio decreases, because as we all know, a smaller sample will definitely bring a higher error. Specifically, as the sampling ratio decreases, the relative error of the queries by using RL-STuner (22.1%) is gradually approaching that of Taster (34.4%), Taster-AW (35.8%), and Taster-FW (32.4%). This is because when the sampling ratio is very low, there might be no sufficient samples to support AQP, so the relative error of queries by using different sample tuning approaches is similar. As shown in Figure 10(b), the average latency of queries by using RL-STuner and Taster-FW decreases as the sampling ratio decreases. However, Taster and Taster-AW have a larger query latency on the 0.1% sampling ratio because they are more focused on the samples which can only be used to answer recent queries. Hence, the system needs more time to access the underlying dataset to get the results for other queries.



(a) Query accuracy (b) Query latency

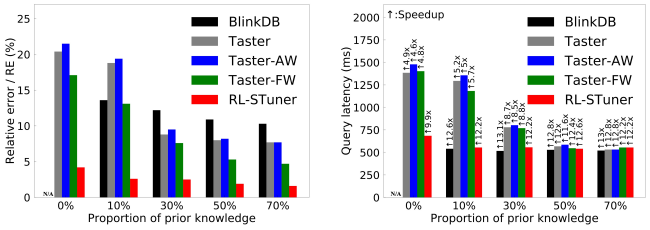
Fig. 11: Performance with different storage budgets on Flights dataset

Effect of storage budget. We vary the storage budget of the sample buffer and conduct experiments on the Flights dataset to evaluate the effect of storage budget on the performance. We configure the storage budget as 30%, 50%, 70%, and 100% of the underlying data size. As shown in Figure 11(a), the average relative error of queries by using the four sample tuning approaches decreases as the storage budget increases, because the system can store more active samples to answer queries. The same reason causes the average latency of queries decreases along with the increase of the storage budget, as shown in Figure 11(b). Specifically, the average latency of



(a) Query accuracy (b) Query latency

Fig. 12: Performance with different time budgets for sample tuning on Flights dataset



(a) Query accuracy (b) Query latency

Fig. 13: Performance with different proportions of prior knowledge on Flights dataset

queries by using the four sample tuning approaches does not decrease significantly as the storage budget increases, since the queries can only be answered by one specific sample.

Effect of tuning time budget. We vary the time budget for sample tuning and conduct experiments on the Flights dataset to evaluate the effect of sample tuning cost on the performance. We configure the tuning time budget as 60 seconds, 30 seconds, and 10 seconds. Fig 12(a) shows the average relative error of the queries in the exploratory workload with various tuning time budgets. As the tuning time budget decreases from 60 seconds to 10 seconds, the average relative error of queries by using Taster, Taster-AW, and Taster-FW increases from 20%/21%/17% to 82%/88%/79%, respectively. This is because these approaches can easily exceed the given time budget due to the high tuning cost and fail to tune the samples in time so that the queries can only be executed on stale samples resulting in an increase in relative errors. On the contrary, RL-STuner can agilely adapt to the changing workload because of its relatively small tuning cost and evaluate queries on tuned samples at most times rather than on stale samples. As shown in Figure 12(a), even when the time budget is 10s, RL-STuner still can achieve a high accuracy. The same reason causes RL-STuner outperforms other tuning approaches on query latency, as shown in Figure 12(b). Specifically, the average latency of queries using RL-STuner on the small tuning time budget setting (10s) is still less than 1 second.

Effect of utilizing prior knowledge. In the above experiments, we demonstrate and discuss the results without any prior knowledge of the workload. However, in some application scenarios, users might be able to provide some prior knowledge about their workload. To evaluate the effect of prior knowledge on performance, we conduct a set of experiments by providing different amounts of prior knowledge. Furthermore, since prior knowledge can also be used for the well-known offline AQP engine BlinkDB [7], we add BlinkDB as one of the baselines in these experiments. Note that the sample tuning in BlinkDB is also triggered when the relative error is larger than the given threshold. We randomly select a proportion of queries from the exploratory workload on the Flights dataset and provide them to BlinkDB, Taster, Tastwer-AW, Taster-FW, and RL-STuner as a priori knowledge of the workload.

Figure 13(a) shows the average relative error of the queries given different proportions of prior knowledge. Since BlinkDB needs a prior knowledge to pre-compute stratified samples, BlinkDB cannot answer the queries when the proportion of prior knowledge is 0% (denoted as N/A). As shown in Figure 13(a), the more prior knowledge given, the higher query accuracy the five approaches can achieve. Obviously, these sample tuning approaches can leverage the prior knowledge to facilitate their performance. There are two reasons for this phenomenon. The first one is that these approaches can learn more knowledge from the prior knowledge than before, thus they can get a better tuning result by "foreseeing" future queries. The second one is that this prior knowledge can make the five approaches select a better active sample set to support the queries in the early stage of exploration. We also find that BlinkDB has a larger relative error than Taster, Taster-AW, Taster-FW, and RL-STuner when the proportion of prior knowledge is high. According to the design of BlinkDB, it will generate samples in terms of the given storage budget and the prior knowledge of the workload [7]. However, due to the constrained storage budget of the sample buffer, BlinkDB might select a sub-optimal sample set in some cases. The same reason causes the average latency of queries to decrease along with the increase of the proportion of prior knowledge, as shown in Figure 13(b).

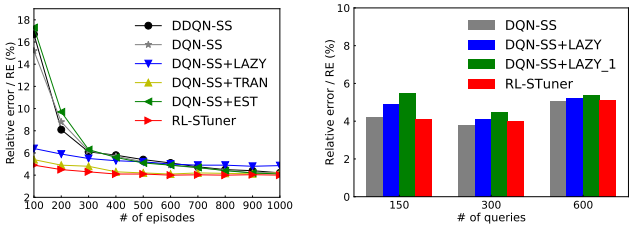
C. Evaluation of Optimization Mechanisms

In this paper, we propose a lazy sample tuning strategy, a parameter transfer approach and a utility estimator to reduce the sample tuning cost. To evaluate the effectiveness of the three optimization mechanisms, we conduct a set of ablation experiments on the Flights dataset to compare the performance on query accuracy, query latency, speedup, and time cost of sample tuning. We can combine the DQN-based sample selection approach (DQN-SS) with the lazy sample tuning strategy (LAZY), the parameter transfer approach (TRAN) and the utility estimator (EST). To evaluate the effectiveness, we compare five combinations: DQN-SS, DQN-SS + LAZY, DQN-SS + TRAN, DQN-SS + EST, and RL-STuner (DQN-SS + LAZY + TRAN + EST). Specifically, for the combinations with TRAN, the number of episodes in the DQN model is set at 500. Otherwise, it is set at 1000. Moreover, we also implement a new approach DDQN-SS with the double DQN model [25] to evaluate the performance using different deep reinforcement learning models.

Table II shows the comparison results of these combinations. As shown in Table II, the performance of DDQN-SS on different metrics is similar to that of DQN-SS since

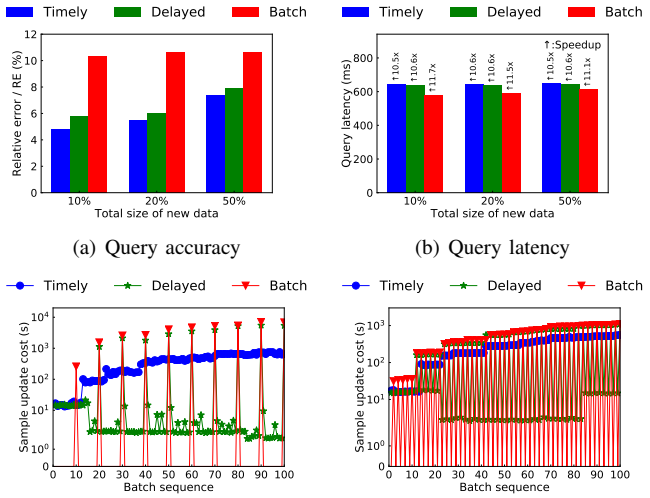
TABLE II: Performance of our sample tuning approach with different deep reinforcement learning models and optimization mechanisms (**LAZY**: the lazy sample tuning strategy, **TRAN**: the parameter transfer approach, **EST**: the utility estimator)

Approaches	Online Query Processing			Offline Sample Tuning
	Relative error (%)	Query latency (ms)	Speedup	Tuning cost (s)
DDQN-SS	4.18	686	9.9x	152
DQN-SS	4.17	688	9.9x	150
DQN-SS + LAZY	4.87	699	9.7x	67
DQN-SS + TRAN	4.16	683	9.9x	107
DQN-SS + EST	4.13	695	9.8x	119
RL-STuner (DQN-SS + LAZY + TRAN + EST)	4.14	685	9.9x	21



(a) Convergence speed of different sample tuning approaches (b) Query accuracy of different sample tuning strategies

Fig. 14: Convergence speed of different sample tuning approaches and effect of different sample tuning strategies



(a) Query accuracy (b) Query latency

(c) Update cost upon 10% new data (d) Update cost upon 50% new data

Fig. 15: Performance with data updates on Flights dataset

the overestimation in the sample selection task is not a serious problem, especially when episodes are few. Therefore, we select DQN for reinforcement learning in our approach. Then, we find that taking the optimization mechanisms when tuning samples can reduce the sample tuning cost by about 86% (from 150s to 21s) while achieving a similar query accuracy, query latency, and speedup as DQN-SS. For the LAZY strategy, it reduces the sample tuning cost by about 55% because unnecessary sample selection and tuning operations can be avoided. For the TRAN, it reduces the sample tuning cost by about 29% while guaranteeing the query accuracy, since the DQN model is initialized with a suitable solution whose performance is better than the random initialization. For the EST, it reduces the sample tuning cost by about 21% because we do not need to access the underlying dataset when calculating the utility of samples. Figure 14(a) shows the convergence speed of these approaches. As shown in Figure 14(a), the convergence speed of DQN-SS is also similar to that of DDQN-SS. Taking TRAN when tuning samples can dramatically improve query accuracy's convergence speed. The convergence speed of DQN-SS + LAZY is also fast because the LAZY strategy only tunes a subset of the active samples in the sample buffer.

Furthermore, to evaluate the negative impact caused by the LAZY, we implement an approach DQN-SS + LAZY_1 to simulate the extreme case that it selects only one sample with the lowest utility for expulsion every once. We compare the query accuracy of DQN-SS, DQN-SS + LAZY, DQN-SS + LAZY_1, and RL-STuner on the Flights dataset. In this set of experiments, we vary the number of queries from 150 to 600. As shown in Figure 14(b), DQN-SS + LAZY_1 has a larger relative error than other approaches since it only tunes one sample in the sample buffer, but the difference between DQN-SS + LAZY_1 and DQN-SS is small (less than 2%). We also find that RL-STuner still achieves good performance for scenarios with more than 150 queries.

D. Comparison of Sample Update Strategies

We take the Flights dataset with 100 million tuples as the base data and generate three new datasets with different sizes as the data to update, which include a dataset with 10 million tuples (10% of the base data), a dataset with 20 million tuples (20% of the base data), and a dataset with 50 million tuples (50% of the base data). To simulate data updates, we divide each dataset into 100 batches equally and feed a batch of data to the system every time when a query ends. The data updates will cause sample updates and further affect the performance of RL-STuner. For data updates, we compare the performance of three sample update strategies, i.e., *Timely*, *Delayed*, and *Batch*, discussed in Section III-C. We set the threshold at 1% of the base data size for the *Delayed* and *Batch* update strategies. Furthermore, we use the R* algorithm to update samples. As shown in Figure 15(a), the average relative errors under both the *Timely* and *Delayed* sample update strategies are much lower than that under the *Batch* strategy since the active samples in the sample buffer can be updated timely by taking the *Timely* and *Delayed* sample update strategies. For

the *Batch* strategy, since the samples are not updated in time, the accuracy of some queries will drop sharply. As shown in Figure 15(b), overall, different sample update strategies have less impact on the query latency. The query latency under the *Timely* and *Delayed* strategies is larger than that under the *Batch* strategy because some new samples have a larger size after the data updates. Figure 15(c) and Figure 15(d) show the sample update cost by taking three different strategies upon 10% new data and 50% new data, respectively. As shown in Figure 15(c) and Figure 15(d), the *Delayed* sample update strategy can complete the sample update at a relatively small cost in most cases by postponing sample updates on the sample pool, and at the same time obtain a similar query accuracy as the *Timely* strategy.

V. Related Work

AQP approaches mainly include two categories: *online aggregation* and *sampling-based AQP*. Online aggregation usually takes samples from the dataset by storing the tuples in random order and uses these sampled tuples to answer the queries [26]–[36]. By online aggregation, users can observe the progress of a query by showing iteratively refined approximate answers, and stop the query execution once its result achieves the desired accuracy [31]. Sampling-based AQP has been extensively studied in the last decades. These approaches generate samples by two ways. **One is online sampling that generates samples at runtime for the input query, such as Quickr [9] and index-assisted sampling method [37]. By generating a new specific sample for each query, the online sampling methods can guarantee the sample returned for a query is independent of those returned for all previous queries [38]. However, the cost of online sampling will increase query latency, and the sample can only benefit one specific query. The online sampling methods ignore the possibility of sample reusing.** Another is offline sampling that draws samples from the underlying data in a pre-processing step according to a priori knowledge of the workload and leverages these pre-computed samples to answer queries [7], [8], [10]–[12], [39]–[47]. Compared to the online sampling approaches, the offline sampling approaches achieve a significant reduction in query latency by reusing samples to answer queries. However, the IDE workload can hardly be predicted since the user usually explores the interest of data by changing queries continuously.

In the past few years, several machine learning-based AQP approaches have emerged [48]–[55]. For example, to achieve higher accuracy for the given query, [55] uses the reinforcement learning model to decide how much budget should be allocated to each sampler when generating a sample. The reinforcement learning model has also been used in DBMS, such as database tuning [56], [57] and materialized view selection [58], [59]. Although the materialized view can provide accurate query results, it requires a large storage space and depends on a static workload.

VI. Conclusion

In this paper, we propose a deep reinforcement learning-based sample tuner *RL-STuner* to make an AQP system able to adapt to the changing workload for IDE. *RL-STuner* uses a DQN model to tune samples from a global perspective. In addition, we propose a set of optimization mechanisms to systematically reduce the sample tuning cost in *RL-STuner*. Extensive experiments on real-world and synthetic datasets show that *RL-STuner* can outperform existing sample tuning approaches on multiple performance metrics.

Acknowledgment

This work was supported by the National Natural Science Foundation of China No.62072113. The authors would like to thank the anonymous reviewers for their constructive comments.

References

[1] K. Dimitriadou, O. Papaemmanouil, and Y. Diao, "Explore-by-example: an automatic query steering framework for interactive data exploration," in *SIGMOD*, 2014, pp. 517–528.
[2] P. Eichmann, E. Zgraggen, C. Binnig, and T. Kraska, "Idebench: A benchmark for interactive data exploration," in *SIGMOD*, 2020, pp. 1555–1569.
[3] Z. Liu and J. Heer, "The effects of interactive latency on exploratory visual analysis," *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 12, pp. 2122–2131, 2014.
[4] R. B. Miller, "Response time in man-computer conversational transactions," in *AFIPS*, 1968, pp. 267–277.
[5] B. Mozafari, "Approximate query engines: Commercial challenges and research opportunities," in *SIGMOD*, 2017, pp. 521–524.
[6] S. Chaudhuri, G. Das, and V. R. Narasayya, "A robust, optimization-based approach for approximate answering of aggregate queries," in *SIGMOD*, 2001, pp. 295–306.
[7] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "Blinkdb: queries with bounded errors and bounded response times on very large data," in *EuroSys*, 2013, pp. 29–42.
[8] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang, "Sample + seek: Approximating aggregates with distribution precision guarantee," in *SIGMOD*, 2016, pp. 679–694.
[9] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding, "Quickr: Lazily approximating complex adhoc queries in bigdata clusters," in *SIGMOD*, 2016, pp. 631–646.
[10] A. Galakatos, A. Crotty, E. Zgraggen, C. Binnig, and T. Kraska, "Revisiting reuse for approximate query processing," *PVLDB*, vol. 10, no. 10, pp. 1142–1153, 2017.
[11] Y. Park, B. Mozafari, J. Sorenson, and J. Wang, "Verdictdb: Universalizing approximate query processing," in *SIGMOD*, 2018, pp. 1461–1476.
[12] T. D. Nguyen, M. Shih, S. S. Parvathaneni, B. Xu, D. Srivastava, and S. Tirthapura, "Random sampling for group-by queries," in *ICDE*, 2020, pp. 541–552.
[13] M. Olma, O. Papapetrou, R. Appuswamy, and A. Ailamaki, "Taster: Self-tuning, elastic and online approximate query processing," in *ICDE*, 2019, pp. 482–493.
[14] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. M. VanBriesen, and N. S. Glance, "Cost-effective outbreak detection in networks," in *SIGKDD*, P. Berkhin, R. Caruana, and X. Wu, Eds., 2007, pp. 420–429.
[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.
[16] Z. Wu, Y. Jing, Z. He, C. Guo, and X. S. Wang, "POLYTOPE: a flexible sampling system for answering exploratory queries," *World Wide Web*, vol. 23, no. 1, pp. 1–22, 2020.
[17] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.
[18] M. A. Wiering and M. van Otterlo, Eds., *Reinforcement Learning*, ser. Adaptation, Learning, and Optimization, 2012, vol. 12.
[19] S. Lohr, *Sampling : design and analysis*. Brooks/Cole, 2010.
[20] K. Li and G. Li, "Approximate query processing: What is new and where to go? - A survey on approximate query processing," *Data Sci. Eng.*, vol. 3, no. 4, pp. 379–397, 2018.
[21] Y. Tao, X. Lian, D. Papadias, and M. Hadjieleftheriou, "Random sampling for continuous streams with arbitrary updates," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 1, pp. 96–110, 2007.

[22] H. Zhang, Y. Zhang, Z. He, Y. Jing, K. Zhang, and X. S. Wang, "An agile sample maintenance approach for agile analytics," in *ICDE*, 2020, pp. 757–768.

[23] *Bureau of transportation statistics.*, Flights dataset, 2020.

[24] *TPC-H Benchmark.*, http://www.tpc.org/tpch/.

[25] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *AAAI*, 2016, pp. 2094–2100.

[26] J. M. Hellerstein, P. J. Haas, and H. J. Wang, "Online aggregation," in *SIGMOD*, 1997, pp. 171–182.

[27] P. J. Haas, "Large-sample and deterministic confidence intervals for online aggregation," in *SSDBM*, 1997, pp. 51–63.

[28] P. J. Haas and J. M. Hellerstein, "Ripple joins for online aggregation," in *SIGMOD*, 1999, pp. 287–298.

[29] S. Wu, S. Jiang, B. C. Ooi, and K. Tan, "Distributed online aggregation," *VLDB*, vol. 2, no. 1, pp. 443–454, 2009.

[30] S. Chen, P. B. Gibbons, and S. Nath, "Pr-join: a non-blocking join achieving higher early result rate with statistical guarantees," in *SIGMOD*, 2010, pp. 147–158.

[31] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. Stoica, "G-OLA: generalized on-line aggregation for interactive analysis on big data," in *SIGMOD*, 2015, pp. 913–918.

[32] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "Approxhadoop: Bringing approximations to mapreduce frameworks," in *ASPLOS*, 2015, pp. 383–397.

[33] S. Han, H. Wang, J. Wan, and J. Li, "An iterative scheme for leverage-based approximate aggregation," in *ICDE*, 2019, pp. 494–505.

[34] F. Li, B. Wu, K. Yi, and Z. Zhao, "Wander join and XDB: online aggregation via random walks," *ACM Transactions on Database Systems, TODS*, vol. 44, no. 1, pp. 2:1–2:41, 2019.

[35] ——, "Wander join: Online aggregation via random walks," in *SIGMOD*, 2016, pp. 615–629.

[36] N. Sheoran, S. Chockchowwat, A. Chheda, S. Wang, R. Verma, and Y. Park, "A step toward deep online aggregation," *Proc. ACM Manag. Data*, vol. 1, no. 2, pp. 124:1–124:28, 2023.

[37] Z. Zhao, D. Xie, and F. Li, "Ab-tree: Index for concurrent random sampling and updates," *Proc. VLDB Endow.*, vol. 15, no. 9, pp. 1835–1847, 2022.

[38] Y. Tao, "Algorithmic techniques for independent query sampling," in *PODS*, 2022, pp. 129–138.

[39] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy, "The aqua approximate query answering system," in *SIGMOD*, 1999, pp. 574–576.

[40] V. Ganti, M. Lee, and R. Ramakrishnan, "ICICLES: self-tuning samples for approximate query answering," in *VLDB*, 2000, pp. 176–187.

[41] B. Babcock, S. Chaudhuri, and G. Das, "Dynamic sample selection for approximate query processing," in *SIGMOD*, 2003, pp. 539–550.

[42] C. Jermaine, "Robust estimation with sampling and approximate pre-aggregation," in *VLDB*, 2003, pp. 886–897.

[43] S. Chaudhuri, G. Das, and V. R. Narasayya, "Optimized stratified sampling for approximate query processing," *ACM Trans. Database Syst.*, vol. 32, no. 2, p. 9, 2007.

[44] L. Sidirourgos, M. L. Kersten, and P. A. Boncz, "Sciborq: Scientific data management with bounds on runtime and quality," in *CIDR*, 2011, pp. 296–301.

[45] K. Li, Y. Zhang, G. Li, W. Tao, and Y. Yan, "Bounded approximate query processing," *IEEE Transactions on Knowledge and Data Engineering, TKDE*, vol. 31, no. 12, pp. 2262–2276, 2019.

[46] X. Liang, S. Sintos, Z. Shang, and S. Krishnan, "Combining aggregation and sampling (nearly) optimally for approximate query processing," in *SIGMOD*, 2021, pp. 1129–1141.

[47] S. Acharya, P. B. Gibbons, and V. Poosala, "Congressional samples for approximate answering of group-by queries," in *SIGMOD*, 2000, pp. 487–498.

[48] Y. Park, A. S. Tajik, M. J. Cafarella, and B. Mozafari, "Database learning: Toward a database that becomes smarter every time," in *SIGMOD*, 2017, pp. 587–602.

[49] Q. Ma and P. Triantafillou, "Dbest: Revisiting approximate query processing engines with machine learning models," in *SIGMOD*, 2019, pp. 1553–1570.

[50] B. Walenz, S. Sintos, S. Roy, and J. Yang, "Learning to sample: Counting with complex queries," *Very Large Data Base Endowment, VLDB*, vol. 13, no. 3, pp. 390–402, 2019.

[51] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig, "Deepdb: Learn from data, not from queries!" *Very Large Data Base Endowment, VLDB*, vol. 13, no. 7, pp. 992–1005, 2020.

[52] A. M. Shanghooshabad, M. Kurmanji, Q. Ma, M. Shekelyan, M. Almasi, and P. Triantafillou, "Pgmjoins: Random join sampling with graphical models," in *SIGMOD*, 2021, pp. 1610–1622.

[53] S. Thirumuruganathan, S. Hasan, N. Koudas, and G. Das, "Approximate query processing for data exploration using deep generative models," in *ICDE*, 2020, pp. 1309–1320.

[54] Q. Ma, A. M. Shanghooshabad, M. Almasi, M. Kurmanji, and P. Triantafillou, "Learned approximate query processing: Make it light, accurate and fast," in *CIDR*, 2021.

[55] J. Peng, B. Ding, J. Wang, K. Zeng, and J. Zhou, "One size does not fit all: A bandit-based sampler combination framework with theoretical guarantees," in *SIGMOD*, 2022, pp. 531–544.

[56] G. Li, X. Zhou, S. Li, and B. Gao, "Qtune: A query-aware database tuning system with deep reinforcement learning," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 2118–2130, 2019.

[57] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, M. Ran, and Z. Li, "An end-to-end automatic cloud database tuning system using deep reinforcement learning," in *SIGMOD*, 2019, pp. 415–432.

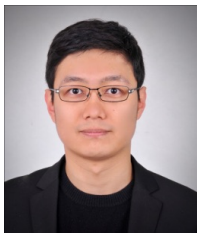
[58] X. Liang, A. J. Elmore, and S. Krishnan, "Opportunistic view materialization with deep reinforcement learning," *CoRR*, vol. abs/1903.01363, 2019.

[59] H. Yuan, G. Li, L. Feng, J. Sun, and Y. Han, "Automatic view generation with deep learning and reinforcement learning," in *ICDE*, 2020, pp. 1501–1512.

Hanbing Zhang received the Ph.D. degree in software engineering from Fudan University in 2022. He is currently a postdoc research fellow in the School of Computer Science, Fudan University. His research interests include interactive data exploration, approximate query processing, and query optimization.

Yinan Jing received the Ph.D. degree in computer science from Fudan University, Shanghai, China in 2007. He is an Associate Professor with the School of Computer Science at Fudan University. He was also a Visiting Scholar with the Department of Computer Science at the University of Southern California. His current research interests include big data analytics, database systems, spatial and temporal data management, and data security and privacy. He is a member of ACM and IEEE.

Zhenying He received the BS, MS, and Ph.D. degrees in computer science from the Harbin Institute of Technology, China, in 1998, 2000, and 2006, respectively. Currently, he is an associate professor in the School of Computer Science, Fudan University. His current research interests include keywords search on structured data, query processing on RDF data, and big data.



Kai Zhang received the Ph.D. degree from the University of Science and Technology of China in 2016. He is an associate professor with the School of Computer Science and Engineering, Fudan University. He was a research fellow with the National University of Singapore from 2016 to 2017. He was a visiting scholar with the Ohio State University from 2013 to 2015. His research interests are mainly in the fields of parallel and distributed computing and database systems.

X.Sean Wang (Senior Member, IEEE) received the Ph.D. degree in computer science from the University of Southern California. He is a distinguished professor with the School of Computer Science, Fudan University, Shanghai, China. Before joining Fudan University in 2011, he was the dorothean chair professor in Computer Science with the University of Vermont. His research interests include data systems and data security. He is the fellow of CCF and the member of ACM.