LearnedSQLGen: Constraint-aware SQL Generation using Reinforcement Learning

Lixi Zhang
Department of Computer Science, Tsinghua University
zhanglx19@mails.tsinghua.edu.cn

Chengliang Chai
Department of Computer Science, Tsinghua University
ccl@tsinghua.edu.cn

Xuanhe Zhou
Department of Computer Science, Tsinghua University
zhouxuan19@mails.tsinghua.edu.cn

Guoliang Li
Department of Computer Science, Tsinghua University
liguoliang@tsinghua.edu.cn

ABSTRACT

Many database optimization problems, e.g., slow SQL diagnosis, database testing, optimizer tuning, require a large volume of SQL queries. Due to privacy issues, it is hard to obtain real SQL queries, and thus SQL generation is a very important task in database optimization. Existing SQL generation methods either randomly generate SQL queries or rely on human-crafted SQL templates to generate SQL queries, but they cannot meet various user specific requirements, e.g., slow SQL queries, SQL queries with large result sizes. To address this problem, this paper studies the problem of constraint-aware SQL generation, which, given a constraint (e.g., cardinality within [1k,2k]), generates SQL queries satisfying the constraint. This problem is rather challenging, because it is rather hard to capture the relationship from query constraint (e.g., cardinality and cost) to SQL queries and thus it is hard to guide a generation method to explore the SQL generation direction towards meeting the constraint. To address this challenge, we propose a reinforcement learning (RL) based framework `LearnedSQLGen`, for generating queries satisfying the constraint. `LearnedSQLGen` adopts an exploration-exploitation strategy that exploits the generation direction following the query constraint, which is learned from query execution feedback. We judiciously design the reward function in RL to guide the generation process accurately. We also integrate a finite-state machine in our model to generate valid SQL queries. Experimental results on three benchmarks showed that `LearnedSQLGen` significantly outperformed the baselines in terms of both accuracy (30% better) and efficiency (10-35×).

1 INTRODUCTION

Many database optimization problems require a large volume of SQL queries, e.g., slow SQL diagnosis [14, 37], database testing [39, 41, 53], optimizer tuning [9, 12, 15, 56, 57], learned cardinality estimator [16, 27]. For example, to make database optimizer more robust, it is important to feed the optimizer with a huge number of SQL queries. For another example, to train a high-quality learned cardinality estimator, it also requires to generate a large number of SQL queries [16]. However, it is rather hard to obtain a large number of real SQL queries due to privacy issues, and thus SQL generation is a very important task in database optimization [12].

Although there are some SQL generation tools, e.g., SQLsmith [39] and RAGs [41], they have several limitations. First, they randomly generate SQL queries and the generated queries may be useless, e.g., empty result. Second, they cannot generate SQL queries that meet user requirements. For example, if we find that an optimizer should be enhanced for optimizing the SQL queries with small cardinality (e.g., cardinality within [1,10]), we need to generate SQL queries satisfying this constraint.

To address this limitation, this paper studies the problem of constraint-aware SQL generation, which, given a constraint (e.g., cardinality within a range, query cost within a range), generates SQL queries satisfying this constraint. A straightforward method, which first randomly generates SQL queries with existing tools [39], and then validates whether each generated SQL query satisfies the constraint, is rather expensive, because each generated query has very low probability satisfying the constraint. Although there are some works [10, 31] that attempt to generate SQL queries satisfying a constraint, they have several limitations. First, they require database experts to craft some high-quality SQL query templates (SQL structures without predicate values). Obviously, it is expensive to craft templates for many constraints, and it is hard to craft templates for new databases that the expert is not familiar with. Second, there are many different constraints (e.g., slow SQL queries for various scenarios), and using the crafted templates (1) may not find queries satisfying the constraints and (2) may miss important SQL queries. Because the experts may miss some important templates, and thus the corresponding SQL queries will be missed.

There are three main challenges in constraint-aware SQL query generation. (C1) SQL queries may have diverse cardinality/cost, and it is challenging to capture the relationship from a constraint to SQL queries. (C2) There are many different constraints required by users in different scenarios, and it is challenging to adapt to different constraints. (C3) The generated SQL queries must be valid (syntactically and semantically correct), and we need to guarantee the query validity. To address these challenges, we propose a system, `LearnedSQLGen`, which utilizes reinforcement learning (RL) to generate queries with target constraints. RL is a typical machine learning paradigm that an agent learns from the query-execution feedback by trial-and-error interactions with the environment (database system), which adopts an *exploration-exploitation* strategy.

SIGMOD'22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3183743

LearnedSQLGen generates queries by trying different actions (*i.e.*, tokens like reserved words, metadata, operands in a query), and the database system (*i.e.*, environment) can return the feedback by executing the generated query. Based on the feedback, LearnedSQLGen can *exploit* the optimal generation direction that leads to the target constraint (addressing C1). Moreover, we utilize a meta-critic network that makes our framework generalizable to other SQL generation tasks with different constraints effectively (addressing C2). Furthermore, we use a finite-state machine to restrict the action space so that each generated query is valid (addressing C3).

To summarize, we make the following contributions.
(1) We propose a reinforcement learning based framework to generate queries with target constraints. To the best of our knowledge, this work is the first attempt that uses a learning-based method to address the constraint-aware SQL generation problem, satisfying cost or cardinality constraints (see Section 3).
(2) We judiciously design reward functions in RL to guide the generation process accurately. For each type of constraint (point or range), we design reward functions, and then we use the actor-critic network to achieve robust training in RL (see Section 4).
(3) We adopt a finite-state machine that incorporates SQL grammar and semantic rules to guarantee query validity, which can be extended to support various types of queries, like nested queries, insert/update/delete queries, etc. (see Section 5).
(4) We design a meta-critic strategy that first pre-trains a model for different constraints and then uses the pre-trained model to support online query generation requirements (see Section 6).
(5) We have conducted experiments on three datasets, and the results showed that our method significantly improved the accuracy and efficiency by 30% and 10-35× respectively (see Section 7).

2 PRELIMINARY

2.1 Problem Formulation

Given a database, a user wants to generate a set of SQL queries that satisfy a constraint. An intuitive question is how to formulate the constraint. Two widely-used metrics for SQL queries are cardinality and cost. The former quantifies the number of results for a SQL query, and the latter quantifies the cost of executing the SQL query. So we use cardinality or cost to formulate the constraint and define the constraint-aware SQL generation problem as below.

Definition 1 (SQL Query Generation). *Consider a database* $\mathbf{D} = \{\mathbf{R_1}, \mathbf{R_2}, ..., \mathbf{R_d}\}$ *with d relations, where each relation has multiple attributes. Given a cardinality/cost constraint* $\mathbf{C}$ *(a point or range constraint) and the number N of generated queries provided by the user, the query generation problem is to generate a set* $\mathcal{Q}$ *of SQL queries, such that* $\forall Q \in \mathcal{Q}$, Q *satisfies the constraint* $\mathbf{C}$ *and* $|\mathcal{Q}| = N$.

Example 1. *As shown in Figure 1, database* $\mathbf{D}$ *has two relations. A user provides a range constraint* `Cardinality in [1K, 2K]` *with respect to the cardinality, which means that the cardinalities of the generated queries should be between 1K and 2K. Also, a user can input a point constraint* `Cost = 10`*, which represents that the user expects the costs of generated queries are close to 10 as much as possible.*

Complexity Analysis. Our problem is NP-hard, which can be proved using the same proof provided in [10](Theorem 4.1). The reduction is from the subset-sum problem (we are given m numbers $s_1, ..., s_m$, and the goal is to choose a subset of numbers with total sum being exactly s). For each given number s_i in the subset sum instance, we create a relation R with s_i tuples with exact the same attribute values. The observation is that if any such tuple is selected by a query, all s_i tuples should be selected. Hence, generating an SQL whose output cardinality is exactly s is equivalent to finding a subset of numbers with total sum being s. Note that although R is exponential sized, the authors in [10] show that R can be encoded by a join of several poly-sized relations.

Remark. (1) In most scenarios, only the database tables associated with schemes are available before SQL query generation. Hence, our framework focuses on the case that there are no available SQL queries. (2) As our framework can well support both the cardinality and cost constraint, we do not distinguish them for ease of representation. (3) We can also allow users to specify the latency as a constraint, but it is sensitive to the hardware environment, so we use cost instead (like optimizers also use cost). (4) Since SQL generation is an NP-hard problem [10], it is rather hard to guarantee that any query or template will not be missed. Thus our method may also miss SQL queries, but our method is capable of discovering more satisfied SQL queries through an exploration-exploitation strategy, as discussed and evaluated in the following sections.

2.2 Related Work

SQL query generation. Existing query generation methods can be broadly divided into two categories.

(1) *Query-driven method.* It relies on a large number of given SQL queries to train a model, and then uses the model to generate similar SQL queries. Liu et al. [26] proposed a GAN-based model to randomly synthesize new SQL queries from historical queries. Hence it cannot be directly applied to our problem for two reasons. First, it requires many prepared training SQLs, which are expensive to acquire in real scenarios. Second, they purely generate random SQLs, and cannot meet the user-specified constraints.

(2) *Query generation without given queries.* Generally speaking, this category can further be classified into two categories. (*i*) Random methods [39, 41]. Slutz et al [41] proposed to generate SQL queries by randomly walking on a parse tree and executing them on multiple database instances so as to compare their results for consistency detection (differential testing). SQLsmith [39] is a typical random query generation tool, which generates complex SQL queries but they may produce empty results. Bati [8] proposed a genetic algorithm that randomly synthesizes queries (e.g., addition/removal of predicates) so as to trigger rarely covered code paths (e.g., spill to disk when a join does not fit in the memory). However, they ignore the important constraints for database testing (e.g., cardinality ranges), and thus have low efficiency in generating desired queries with constraints. (*ii*) Template-based methods [10, 31]. They rely on some given SQL templates and change the values in the predicates of the templates to generate queries satisfying the given constraints. Bruno et al [10] proposed to generate desired queries by tweaking the predicate values in the given query template (e.g., the `x` in `R.a < x`). To satisfy the cardinality constraint, they utilize a hill-climbing algorithm that selects predicate values to minimize the distance from the target constraint. Besides, Chaitanya et al [31] proposed a space-pruning technique to reduce the searching space on the SQL templates. Specifically, they iteratively sample some values of the predicates and restrict the search space based on the top-k selected

areas with shortest distance from the cardinality. Template-based methods have higher efficiency than random methods. However, (i) the generation performance heavily depends on the quality of templates, which need to be manually designed by experts; (ii) Although some high-quality templates can be provided by experts, it is hard for these templates to cover various constraints thoroughly.

Reinforcement learning is an ML paradigm that an agent learns from the feedback from the environment through trial-and-error interactions. RL is often utilized in sequence generation, which verifies that RL naturally fits our problem because a query can be seen as a sequence of tokens. For example, applications like machine translation [19, 35, 48], text generation [7, 36], and dialogue system [24, 40] can be solved by RL. However, their problems are very different from us, and thus the solutions cannot be applied.

Learning Models for Databases. Recently machine learning has gained great development and received widespread attention in the database community [5, 20–23, 29, 45, 49, 52, 54, 55, 58]. Database researchers have used it in many topics like entity matching [11, 33], approximate query processing (AQP) [28, 38]. For database systems, there are learning-based works like reinforcement learning for knob tuning [6, 22, 52], reinforcement learning for query optimizer [30, 30, 50], hybrid algorithms for materialized view selection [4, 51], and deep learning for learned data layouts [13, 18, 34]. These works focus on optimizing database components. Note that many query-based methods (e.g., cardinality estimation, index/view advisor) [22, 30, 30, 50, 51, 58] rely on numerous queries to train learning models. Here we try to solve the query generation problem with reinforcement learning.

3 SYSTEM FRAMEWORK

We propose `LearnedSQLGen`, a query generation framework using an RL model. In this section, we first summarize the challenges and overall solution of `LearnedSQLGen` (Section 3.1), and then introduce the training and inference step respectively at a high level in Section 3.2 and 3.3. Note that in this section, we introduce the overall framework that generates queries using the RL model for one given constraint, and will discuss how to generalize the model to multiple different constraints in Section 6.

3.1 Overview of LearnedSQLGen

Although there are many cost-estimation approaches to evaluate the cardinality and cost of queries, they cannot directly generate queries that satisfy a constraint, because they have to enumerate many queries to check whether the queries satisfy the constraint and obviously this solution is rather expensive, as most of the generated queries cannot satisfy the constraint.

Basic Idea of `LearnedSQLGen`. We propose to use a reinforcement learning (RL) framework to address the constraint-aware SQL generation problem. RL is an ML paradigm that an agent learns from the feedback through trial-and-error interactions. Hence, there is a natural connection between RL and our problem. More concretely, considering the current state (i.e., current generated query), the agent predicts the next optimal action (i.e., the next token), which hopefully can lead to the highest reward, i.e., satisfying the constraint. Thus, the RL mechanism seamlessly fits the generation process of SQL queries with constraints, and thereby it can be definitely used to solve our problem. Furthermore, the reason why RL can well solve our problem is that the exploitation of RL guarantees the accuracy of generated queries (*i.e.*, meeting the constraints), and the exploration allows us to generate more queries with large diversity rather than generating highly similar ones towards existing directions. To summarize, our RL-based method adopts an *exploration-exploitation* strategy, to learn a policy by utilizing execution feedback (e.g., whether a generated SQL satisfies the constraint), in order to guide the query generation process towards meeting the constraint.

There are three challenges $w.r.t.$ our problem.

(1) Generate valid SQL queries by guaranteeing syntactic and semantic correctness. We define a finite-state machine (FSM) that incorporates predefined grammar patterns and semantic restrictions to guarantee the correctness of queries. We will briefly discuss this in Section 3.2 and introduce the details in Section 5.

(2) Guide the generation direction to meet the constraints by computing the expectation of SQL queries. We carefully design the reward in the RL framework to guide the generation. For example, generating a query satisfying the constraint will be assigned a high reward, which is the "exploitation" strategy that makes the generation close to our expectation (see Section 4 for details).

(3) Generate multiple different queries satisfying the constraint. Obviously, the user definitely wants diverse queries rather than almost the same ones. To this end, we use the policy-based RL framework that chooses the actions probabilistically, which can explore more query spaces that possibly meet the query constraint, *i.e.*, the "exploration" strategy (see Section 4 for details).

In a nutshell, leveraging the FSM and the exploration-exploitation RL framework, `LearnedSQLGen` can generate valid and diverse queries that meet user's constraints. As below, we will summarize the overall workflow of `LearnedSQLGen`.

`LearnedSQLGen` *workflow.* The `LearnedSQLGen` framework is shown in Figure 1. It first trains an RL model based on the database **D** and cardinality/cost constraint **C**. The model takes as input a query (including subquery), *i.e.*, *state*, and computes the optimal token (e.g., reserved words, table/attribute name, predicate value) to be added to the query, *i.e.*, *action*. Then given the model and **D**, in the inference step, we can generate queries that satisfy the constraint using the model. Next, we will overview the above steps.

3.2 Training

We model the query generation problem as a sequential decision making process which can be potentially solved by the RL model. We define the basic traits of RL in our system for training as follows. ***State*** is represented as a sequence of tokens of the current generated SQL query. Basically, there are two kinds of queries during generation. (1) *Complete generated queries.* If a query encounters an `EOF` token, it is a complete generated query that will not be further extended. Then it will be sent to the environment for execution, and the feedback will be used as the reward to guide the training process. Note that our FSM guarantees that the query is valid and executable. (2) *Partial queries.* If a query has not met the `EOF`, it is a partial query that should be further extended by tokens. Note that some partial queries are also executable. In our framework, both partial executable queries and complete queries will be executed for training. All above queries should be encoded as a state.

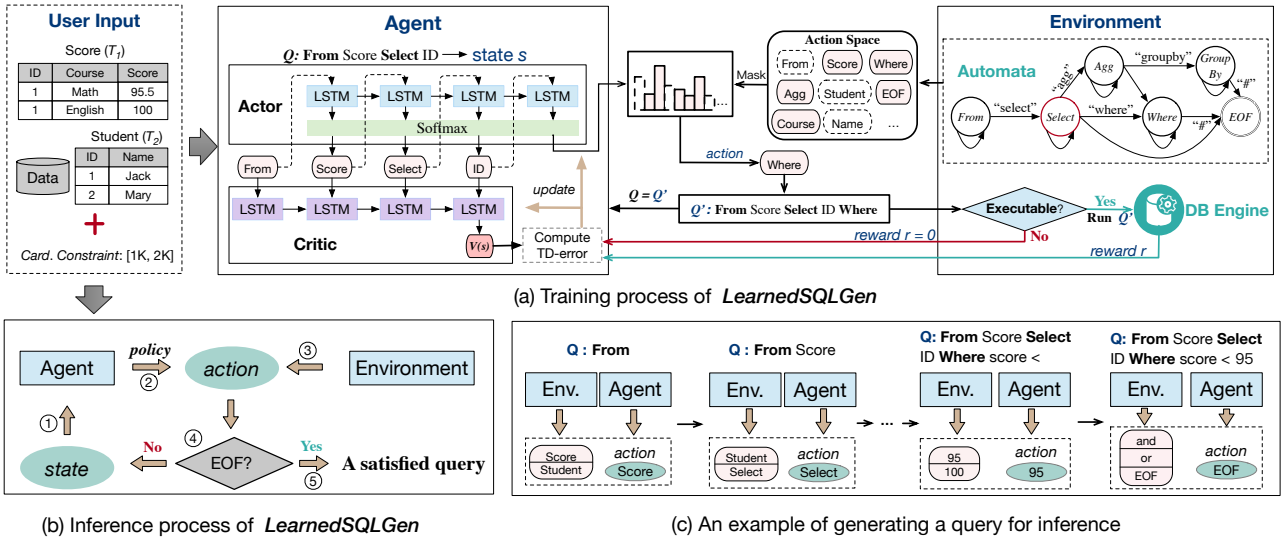


Figure 1: The LearnedSQLGen Architecture.

Action is the next token to select, which is classified into 5 types: *i*). Reserved words in SQL grammar (*e.g.*, `Select`, `Where`); *ii*). Meta data of the schema (*e.g.*, T_1, T_2); *iii*). Cell values sampled from each table in the database (*e.g.*, 95.5, 100); *iv*). The operator (*e.g.*, >, =); *v*). `EOF`, denotes the query is completely generated. Hence, given the database **D**, the action space is fixed (denoted by $\mathcal{A}$), but not all actions in the space can be selected considering the query validity, which is controlled in the environment (see Section 4).

Environment plays two roles in `LearnedSQLGen`. On the one hand, based on the current generated query, it uses FSM to prune the action space for query validity. On the other hand, after each action is applied, a query is generated. If it is executable, the environment will return the estimated cardinality, based on which a reward is generated to guide the training process (see Section 5).

Reward is returned by the environment, which is used to update the policy network, so as to guide the generation process towards satisfying the constraint. A high reward denotes that the generated query satisfies the constraint **C**, and vice versa. But during the generation process of each query, if a partial query is not executable, the reward is returned as 0.

Agent considers the current generated SQL (i.e., current state), and chooses an action based on the learned policy π_θ. In `LearnedSQLGen`, our agent leverages the *actor-critic* method [25], which consists of an *actor* network and a *critic* network. The former is utilized to choose the action based on π_θ, and the latter is used to update the policy based on the reward (see Section 4).

Next, we show the overall training process in Figure 1(a), considering the interaction between the above components.

Overall training process. Initially, as shown in Algorithm 1, the training process takes as input the constraint **C**, and database **D**, based on which the entire action space $\mathcal{A}$ is fixed. Then the training step starts. Note that we do not need training queries in advance, and instead, we simultaneously generate training queries and train the model. That is, we propose to generate queries using the RL model, and leverage these queries associated with their estimated cardinality as the training data to improve the model. Then after training, a number of satisfied queries are generated.

Specifically, each training query is generated from scratch (Line 2) in a token-by-token manner, and here we illustrate our training process from a partial query, *i.e.*, Q = "". Then Q is represented to a *state* s_k=('From', 'Score', 'Select', 'ID') (Line 4), where k is just used to identify different states. Next, the *agent* (*i.e.*, the actor network) takes as input s, and computes the probability of each *action*, *i.e.*, $\pi_\theta(a|s) = p(a|s, \theta)$, $a \in \mathcal{A}$. Hopefully, the higher the probability is, the larger long-term reward of the corresponding action is expected to acquire if the action (*i.e.*, the next token) is selected, which means that the finally generated query is more likely to satisfy the constraint.

As the action space is large, and most actions in $\mathcal{A}$ will lead to an invalid query, so an FSM in the *environment* helps to prune the action space and guarantee the validity (Line 5). For instance, given Q, the FSM masks the actions like `From`, `Student`, `Name`, etc. Hence, the policy samples from the rest unmasked ones based on the probabilities. Suppose that `Where` is selected, so we update Q to " `From Score Select ID Where`" (Line 8). Next, since it is not executable, a reward 0 is returned to the critic network of the *agent* (Line 11). Otherwise, a non-zero reward that reflects how the query satisfies the constraint will be returned. The reward is computed by comparing the estimated cardinality (computed by the cost estimator of databases) of the query with the constraint (Line 10). Note that we do not use the real cardinality for the efficiency issue. Then the critic network estimates the long-term reward of the current state (denoted by $V(s)$), which is together with the returned reward to compute the *temporal-difference error* (TD-error), and updates the two networks for optimized action selection (Line 12).

3.3 Inference

Although the training process can already generate some satisfied queries, if the user requires many more SQL queries, we can directly use the trained model to generate satisfied queries without updating the network, *i.e.*, the inference step. Also, the inference step allows the users to call the trained model to generate queries satisfying

Algorithm 1: LearnedSQLGen Training

Input: Database D, constraint **C**, a user-designed FSM.
Output: A trained RL model M.

```
1  for each episode during training do
2      Set a query Q =""; k = 1 ;
3      while the action a ≠ EOF do
4          State s_k = LSTM(Q);
5          Mask some actions in A based on Q and FSM;
6          Action a_k is sampled based on p(a|s, θ), a ∈ A;
7                  // p(a|s, θ) is output by the model M.
8          Add a_k to Q;
9          if Q is executable then
10             Reward r is the computed based on C and the
               estimated Card./Cost of Q in database;
11         else r = 0;
12         Use the reward to update the network;
13         k = k + 1 ;
14 return M;
```

the constraint **C** at any time, without retraining the model. In other words, the inference is the forward pass of the actor network. The details is illustrated as below.

The overall inference process is as shown in Figure 1(b) and the algorithm of generating queries is shown in Algorithm 2. Each query is generated from scratch token-by-token. Initially, we start from an empty query, which is also regarded as a partial query (Line 1). Given a partial query, it is first represented as a new state (step ①, Line 3). Then the agent takes as input the state, uses the learned policy to compute a probability for each action in $\mathcal{A}$ (step ②, Line 5). Meanwhile, considering the current query, the FSM in the environment masks a number of actions to guarantee the validity (step ③, Line 4). Then, the next action, *i.e.*, the next token, is selected by the policy and FSM, and added to the partial query (step ④, Line 7). The above steps iterate until an action with the `EOF` is selected (step ⑤), and then a query is generated and output (Line 8). If the user wants to generate N queries, we repeat this process for N times.

Example 2. *We suppose that* `From` clause *is a basic and indispensable constitution part of SQL,* `LearnedSQLGen` *starts to generate SQL from it, i.e., the first action (This can be defined by the FSM, where* `From` *can be the start point). Then the agent picks tokens one by one according to well-learned policy π_θ. For example, in Figure 1(c), for the second action, the current state s =* (`'From'`)*. Then based on the FSM, the second action can be selected from {*`'Students'`,`'Score'` *}, which can guarantee a valid query, and the agent selects* `'Score'`*. For another instance, for the 8-th action, current state is* (`'From'`, `'Score'`, `'Select'`, `'ID'`, `'Where'`, `'Grade'`, `'<'`)*, and the next token can be selected among cell values sampled from column score(e.g., 95, 100) while others are masked by the FSM. According to π, the agent picks the token '95' with a high probability. Next, the* `EOF` *is selected as the action, and thus the query is completed.*

Note that the inference step can only generate queries satisfying the constraint **C**. In this case, if a user specifies another different constraint, the previous trained model cannot directly work. Hence, in Section 6, we propose a meta-critic network that leverages the historical training experience to make our method generalize to other constraints efficiently.

Algorithm 2: LearnedSQLGen Inference

Input: Database D and constraint **C**.
Output: A satisfied query Q.

```
1  Q = "";
2  while the action a ≠ EOF do
3      State s_k = LSTM(Q);
4      Mask some actions in A based on Q and FSM;
5      Action a_k is sampled based on p(a|s, θ), a ∈ A;
6              // p(a|s, θ) is output by the model M.
7      Add a_k to Q;
8  return Q;
```

4 RL IN LEARNEDSQLGEN

Our policy in the agent is implemented by the reinforcement learning model. We first introduce the model design for the query generation problem, *i.e.*, the state representation (Section 4.1), reward design (Section 4.2) and the actor-critic networks (Section 4.3).

4.1 State Representation

To utilize the RL model, we need to represent the state, which is the input of model. In our problem, as discussed in Section 3.2, each state corresponds to a query (probably an intermediate one), which consists of a sequence of tokens (*i.e.*, actions). Therefore, we should represent the tokens (actions) first. Although there exist many types of tokens, we can represent them to support complicated queries. Especially for predicates w.r.t. a large number of distinct cell values, it is hard to represent them.

Next, we see how to address the above issues.

Token (action) representation. For ease of training and inference, we map different types of tokens to the same encoding space. First, we introduce the 5 token types.

- Reserved words in the SQL grammar. To be specific, we support {`Select`, `From`, `Where`, `Groupby`, `Having`, `Order BY`, `MAX/MIN`, `Sum`, `AVG`, `Count`, `Exist`, `In`, `and`, `or`, `not`}
- Meta data of the schema including the table and attribute names.
- Cell values in the data.
- The operators. We support {>, =, <, ≥, ≤}.
- `EOF`, indicates that a query is generated completely.

All the tokens above constitute the entire action space $\mathcal{A}$. Overall, we map each of the above tokens to a one-hot encoding, *e.g.*, E(`Select`) → 00000, E(`From`) → 00001, ..., E(`Score`) → 01000 etc. Then the the second challenge arises, *i.e.*, encoding all the cell values (the third type) is sometimes impractical due to the large action space. To this end, we tackle the cell values as follows.

Generally, there are three common types of data (cell values) in database: *numerical*, *categorical* and *string* that we can support.

For *categorical* data, like the attribute `Gender`, we just treat the values the same as the other types because the number of distinct values of *categorical* data is always not large, *i.e.*, E(`Male`) → 00100, E(`Female`) → 00010.

However, *numerical* data may have a large number of distinct values. Hence, to reduce the action space, for each numerical attribute, we randomly sample k values from the attribute before training and encode them to a one-hot vector. Then once we generate a value of this attribute, a value among the sampled k values will be selected by the agent.

For *string* data, we can sample k strings and use them the same as the numerical data, and support $\{=, >, <\}$ for string data.

State (query) representation. The state is naturally represented as a sequence of token (query) representations. To be specific, we use s_t to denote a state, where the subscribe t means that there are t tokens in the sequence, *i.e.*, the query Q_t. For example, the query Q_4 = "`From Score` **`Select`** `ID`" corresponds to the state s_4={E(`From`), E(`Score`), E(`Select`), E(`ID`)}.

4.2 Reward Design

We present the reward design based on the constraint types (*i.e.*, point and range) of the user's input. The essence of the reward is to reflect the difference between the feedback from the database and given constraint. Intuitively, the smaller the difference is, the higher reward should be given to the agent, and vice versa.

Point constraint is in the form of $\mathbf{C}$: `Card` = c or `Cost` = c, where c denotes the user requirement. Given a generated query Q_t, $e_t = 1(0)$ denotes that the query is (not) executable and $\hat{c}_t$ denotes the estimated result of Q_t under database $\mathbf{D}$. we use $\delta_t = min(\frac{\hat{c}}{c}, \frac{c}{\hat{c}})$ as the reward. If c or $\hat{c}$ is zero, we set δ_t as 0.

$$r_t = \begin{cases} \delta_t & e_t = 1 \\ 0 & e_t = 0 \end{cases}$$

Example 3. *Suppose a point constraint,* `Card` = $10,000$*, and the agent generates an executable query with estimated cardinality* $\hat{c}_t = 100$*. Hence a low reward 0.01 is returned to decrease the probability of selecting queries like it again. For another query with the estimated cardinality* $\hat{c}_t = 11,000$*, we give a high reward 0.9 to encourage the agent to explore along this direction.*

Remark. Although we are ultimately interested in the performance(*i.e.*, cardinality or cost) of a completely generated SQL (*i.e.*, a_T =`'EOF'`), simply awarding the end reward after the `'EOF'` while giving zero as intermediate reward results in a sparse training signal for the agent. Therefore, we also give the computed reward if partial queries can be executed. Besides, for all types of queries, we will compute an estimated long-term reward for each of them, which is done by the critic network (see Section 4.3 in detail). Note that the long-term reward is not like r that is directly derived from the environment.

Range constraint is in the form of $\mathbf{C}$: `Card` = $[c.l, c.r]$ or `Cost` = $[c.l, c.r]$, which requests that the cardinalities/costs are in $[c.l, c.r]$. e_t has the same meaning with the point constraint. Different from that, if $\hat{c}_t \in [c.l, c.r]$, we assign a positive reward 1. Otherwise, we consider whether $\hat{c}_t$ is near to the range, so as to assign a proper reward. To be specific, we use $\delta_t^l = min(\frac{\hat{c}}{c.l}, \frac{c.l}{\hat{c}})$, $\delta_t^r = min(\frac{\hat{c}}{c.r}, \frac{c.r}{\hat{c}})$ to denote how close is $\hat{c}$ to the left and right bound of the range respectively. Hence, $max(\delta_t^l, \delta_t^r)$ can be utilized to measure how close is $\hat{c}$ to the range. For ease of representation, we introduce a notation $n_t = 1$ to denote $\hat{c}_t \in [c.l, c.r]$. Naturally, $n_t = 0$ means that $\hat{c}_t < c.l$ or $\hat{c}_t > c.r$.

$$r_t = \begin{cases} 1 & e_t = 1 \ \& \ n_t = 1 \\ max(\delta_t^l, \delta_t^r) & e_t = 1 \ \& \ n_t = 0 \\ 0 & e_t = 0 \end{cases}$$

Example 4. *Suppose a range constraint* `Card` = $[1K, 2K]$*, the agent generates a satisfied query with estimated cardinality* $\hat{c}_t = 1.5K$*, and thus the returned reward is 1. For another query with* $\hat{c}_t = 10k$*, which is higher than the upper bound of the range, we return the corresponding relatively small reward 0.2.*

4.3 LearnedSQLGen via Actor-Critic

Given the state representation, the policy in the agent takes as input the state, and infers the optimal action that should be taken in the next step, which is the core part of the RL model. However, the typical policy network cannot achieve good performance because our cumulative rewards are likely to result in high variance among the returned rewards. Hence, we utilize the actor-critic method to alleviate this issue. Moreover, to avoid generating many same queries, we use a entropy regularization technique to adjust the objective function for generating different queries.

Typical policy-based reinforcement learning method (like the REINFORCE algorithm [46]) always relies on optimizing the parameterized policy with respect to the expected long-term reward. To be specific, the objective of the policy network is to maximize $J(\theta)$ defined as below,

$$J(\theta) = \mathbb{E}_{\pi_\theta}[R(\tau)] = \sum_{\tau} p(\tau|\theta) R(\tau_{1:T}) \tag{1}$$

where $\tau = (s_1, a_1, .., s_T, a_T)$ denotes a trajectory leading to a fully generated query, *i.e.*, a_T = `EOF`. $R(\tau_{1:T}) = \sum_{t=1}^{T} r_t$. Intuitively, to maximize $J(\theta)$, given a high reward $R(\tau)$, we aim to learn a policy to increase the probability of selecting the trajectory τ, *i.e.*, $p(\tau|\theta)$. Given the objective function, the REINFORCE algorithm uses the gradient ascent to update the parameters θ in the direction of the gradient,

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}(\sum_{t=0}^{T} \nabla_\theta log \pi_\theta(a_t|s_t) R(\tau_{t:T})) \tag{2}$$

Motivation of actor-critic networks. As discussed above, the REINFORCE introduces in high variability in cumulative reward values ($R(\tau) = \sum_{t=0}^{T} r_t$) because these trajectories during training can deviate from each other at great degrees, which leads to instability and slow convergence [44]. One way to address this issue is to subtract the cumulative reward by a *baseline*, which intuitively makes smaller gradients, and thus smaller and more stable updates. Therefore, the actor-critic network strategy is proposed [25]. The actor network learns the policy distribution to select an action. And for adjusting the cumulative reward, the critic network is to estimate the *baseline*, which can be regarded as the expected long-term reward under certain state.

Next, we first introduce the forward pass of the actor and critic networks respectively, and then illustrate how to update the networks based on the reward and baseline.

The actor network. Since the Long Short-Term Memory(`LSTM`) [17] is a typical structure to model a sequence, the actor network takes as input the state representation, followed

Algorithm 3: Actor-Critic Training for Query Generation

Input: Database D and constraint $\mathbf{C}$
Output: actor-critic networks parameterized by θ, ϕ
Initialize parameter vectors θ, ϕ;
while *not to the max iterations* **do**
  Sample a batch of trajectories following the policy π_θ
  for *each trajectory* **do**
    Compute $\nabla J(\theta), \nabla \mathcal{L}_\phi$ /* using Eq. 4 */
  $\theta = \theta + \alpha \nabla J(\theta)$
  $\phi = \phi + \beta \nabla \mathcal{L}_\phi$

by an `LSTM` and `Softmax` layer, and outputs the policy distribution, *i.e.*, $\pi_\theta(s_t) = \texttt{Softmax}(\texttt{LSTM}(s_t))$. Then the probability of each action $\pi_\theta(a|s_t)$ can be derived, and the agent samples based on the probability distribution.

The critic network estimates the value function, including the *state-value* (the V value) and *action-value* (the Q value). To be specific, the critic network uses a separate `LSTM` network parameterized by ϕ, and outputs the V value, *i.e.*, $V_\phi(s_t)$, which is the estimation of long-term reward from the state s_t using the policy π. Also, the Q value can be naturally computed by $Q_\phi(s_t, a_t) = r_t + V_\phi(s_{t+1})$ [44]. It denotes the estimation of long-term reward that the action a_t is taken from the state s_t using the policy π.

Training (updating) the actor-critic networks. Recap that in the REINFORCE, the high variance of cumulative rewards in Eq. 2 leads to unstable and inefficient training. To address this, we use the V value as a baseline to be subtracted by the rewards for reducing the variance which is produced by the critic network and it is proved that this does not introduce any bias [44].

To be specific, we use $A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$ to replace R_t in Eq. 2, because R_t is an estimate of $Q(s_t, a_t)$ and $V(s_t)$ is the baseline. Hence, the gradient becomes,

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}(\sum_{t=0}^{T} \nabla_\theta log \pi_\theta(a_t|s_t) A(s_t, a_t)) \tag{3}$$

where $A(s_t, a_t)$ can be estimated by $r_t + V_\phi(s_{t+1}) - V_\phi(s_t)$, named by temporal-difference (TD) error. From another perspective, since the baseline, *i.e.*, the V value, is the expected long-term reward of a state, the TD error reflects the advantages/disadvantages of different actions from the state.

Then we discuss how to update the critic network. To estimate the V value accurately, the difference between $r_t + V_\phi(s_{t+1})$ and $V_\phi(s_t)$ should be as small as possible. Hence, we can find that the TD error can also be used to update the critic. To be specific, the loss function should be written as $\mathcal{L}_\phi = (r_t + V_\phi(s_{t+1}) - V_\phi(s_t))^2$.

Entropy regularization. We can further improve the diversity of generated queries through adding a regularization term ($\mathcal{H}(\pi_\theta(\cdot|s_t))$ to the loss function[32, 47]. It denotes the entropy of the probability distribution of actions given a state. The higher the entropy, with a higher probability that the agent can sample diverse actions rather than keeping to select the most likely one. Specifically, we can rewrite Eq. 3 as,

$$\nabla J(\theta) \approx \sum_{t=0}^{T} [\nabla_\theta log \pi_\theta(a_t|s_t) A(s_t, a_t) + \lambda \nabla_\theta \mathcal{H}(\pi_\theta(\cdot|s_t))] \tag{4}$$

where λ controls the strength of the entropy regularization term.

Overall training algorithm Algorithm 3 summarizes the above proposed process. `LearnedSQLGen` first initializes the parameters (Line 1). In each iteration (Line 2-7), `LearnedSQLGen` generates a batch of queries using the learned policy (Line 3). Then it computes the gradient for each trajectories (Line 4) to update the parameter (Line 6-7). Finally, it outputs a well-trained actor-critic network.

5 FSM IN THE ENVIRONMENT

We introduce the designed finite-state machine (FSM) in the environment, which is utilized to mask some possible actions, and thereby the validity of queries is guaranteed. The FSM is built from the typical SQL grammar and can be extended flexibly by the users, so as to generate various types of queries.

Definition 2. *(Finite-state machine in* `LearnedSQLGen`*). Given a database* $\mathbf{D}$ *with d relations, an FSM in* `LearnedSQLGen` *can be defined as a directed acyclic graph (DAG)* $\mathcal{G}_\mathbf{D}$*, which is utilized to describe the generation scope of each token on a query, and guarantee the validity. In* $\mathcal{G}_\mathbf{D}$*, each node (edge) is represented as an FSM-state (FSM-action). Specifically, each intermediate query during the generation process corresponds to an FSM-state, and each possible token to be added to an intermediate query corresponds to an FSM-action.*

To distinguish the FSM-state (action) from the state (action) in RL (Section 4), we use the node (edge) to denote FSM-state (action) for ease of representation. Next, we illustrate how to build the FSM.

SQL grammar to FSM. Actually, our FSM (Figure 2) can be easily built and extended from the SQL grammar, which are categorized into several cases for ease of representation.

[*Case 1: SPJ Query.*] This case describes the commonly-used `Select-Project-Join` query. The FSM in Figure 1 is a summarization of the one in Figure 2, where each node in Figure 1 corresponds to a more detailed SQL structure described by the grammar.

For example, after the `From`, we consider two categories: (1) query from a table, and (2) query from multiple tables with join. More concretely, the `From` clause is related to the blue nodes of Figure 2, where $n_1 \rightarrow n_2 \rightarrow n_5$ (query from the single table `Score`) and $n_1 \rightarrow n_3 \rightarrow n_7$ (query from `Student`) are built based on the first category. The second category, *i.e.*, $n_1 \rightarrow (n_2$ *or* $n_3) \rightarrow n_6$ refers to joining the two tables. Note that the corresponding join keys will be automatically added. Then, we come to the `Select` clause, which is associated with the green nodes n_5, n_6 and n_7. The contents are specified by the `selectTerm+` in the grammar, including the columns to be projected defined in Case $*$, *e.g.*, `T1.ID`.

[*Case 2: Nested Query.*] As shown in Table **??**, we support nested queries by adding `QUERY` in the `Where` or `From` clause. Then, ideally, subqueries can be generated recursively. Consequently, in the FSM, we add another branch starting with `Select` $\rightarrow n_{14} \rightarrow$ `From` $\rightarrow \cdots$.

[Case 3: Aggregation Query.] We also support aggregation queries by adding the `Groupby` and `Having` clauses, which are also integrated into the FSM.

[*Case 4&5&6: Insert/Update/Delete Query.*] Our FSM can support different query types including insert, update and delete.

Syntactic and Semantic Checking. Our method can support syntactic and semantic correctness checking by integrating rules into FSM. First, syntactic correctness checking verifies that reserved words, object names (table/column), operators, delimiters, and so

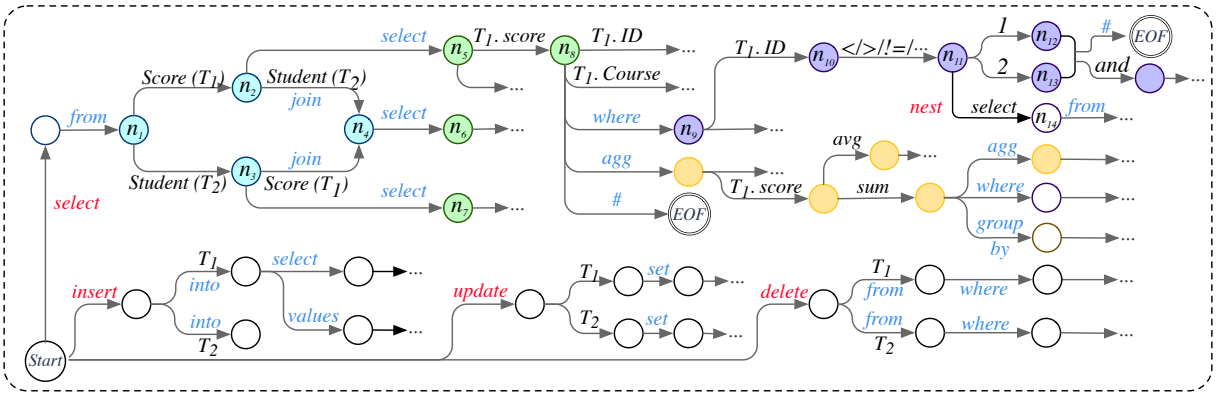


Figure 2: An Example of the FSM in `LearnedSQLGen`.

on are placed correctly in an SQL query. This guarantees to generate valid SQL queries. Second, semantic correctness checking verifies that references to database objects and variables are valid and the datatypes are correct. For example, nonexistent columns/tables cannot be used; only numerical attributes can be included in average/sum/max/min aggregation operations; columns with different datatypes cannot be joined, e.g., `ID` and `Name`.

Meaningful Checking. It is rather hard to support meaningful checking because it relies on domain knowledge. Our current version supports rule-based meaningful checking. For example, if a user defines a rule "two columns can join, only if they have Primary-key-Foreign-key relations or user-specified join relations", then two columns satisfying this rule can be joined. We leave supporting more general meaningful checking as a future work.

Dynamic FSM construction. The FSM may be large. To address this, in fact, we can build it as the query generation process goes on the fly. For example, as shown in Figure 2, after reaching n_1, if the agent selects `Score` rather than `Student`, n_2 is produced and n_3 does not need to appear, and thus many branches can be pruned. Hence, given an intermediate query, we generate the edges based on the SQL grammar and some semantic rules. After the agent selects an edge, we produce a new node while pruning the rest edges. We repeat the above step until the `EOF` is selected.

Example 5. *Given* **D** *with two relations (*`Score` *and* `Student` *in Figure 1), we can build an FSM as shown in Fig 2, which guides the generation of a query. To be specific, it starts from a* `Start` *node, corresponding to a vacant query. Then the* `From` *edge is necessary in a query, leading to the node n_1. Then there are two ways to go, i.e., either selecting* `Score` *or* `Student`*, leading to n_2 or n_3. The selection is done by the agent in the RL framework (see Figure 1). Finally, when we reach the* `EOF` *node, a query is fully generated.*

Supported and Unsupported SQL Syntax. Our current version supports the following SQL syntax: Insert/Delete/Update/Select, Selection, Projection, Join, Aggregation, Group by, Having, Nested Queries. Like is not yet supported. A possible way to support Like is incorporating the keyword "Like" into the FSM and sampling substrings from the values of a column as "values", and we leave the details of this as a future work.

6 PRE-TRAINING FOR DIFFERENT CONSTRAINTS

In the previous sections, given a user-specified constraint, we can train a model to generate queries satisfying the constraint. However, if a users want to generate queries with another constraint, a brute-force method is to retrain a new model so as to obtain an accurate generation result. Hence, it is prohibitively expensive to train a number of models from scratch when there are queries to be generated with various constraints, which is a common phenomenon in reality. To address this goal, a natural solution is to fine-tune trained model corresponding to a distinct range or point constraint, but there exist infinite number of ranges or points in the cardinality/cost domain, and thus we cannot train and store so many models.

Hence, inspired by the meta-critic network [43], we propose to just train a single model on a relatively large cardinality/cost domain. Then given a task with a new constraint, we can efficiently generalize the model to support the new constraint. Next, we introduce the motivation of the meta-critic network, and show how to conduct the training and inference steps of the network.

Motivation of the meta-critic network: meta-critic is a meta-learning method with good generalization ability, which is studied based on the actor-critic network as discussed in Section 4. Specifically, recap that the actor-critic network jointly trains a pair of networks where the actor learns to provide the solution, and the critic learns how to effectively supervise the actor. Different with conventional actor-critic model, the key idea is to learn a meta-critic: a state-value function(*i.e.*, V) neural network that learns to criticize any actor trying to solve any specified task. Intuitively, with multiple actors trains with one shared meta-critic, it learns the transferable knowledge among these actors and tasks that allow actors to be trained efficiently and effectively on a new problem.

Bridging the meta-critic network to our problem: To improve the generalization ability of our method, we will apply the meta-critic network to our problem. Note that the meta-critic network takes as input different but related tasks. In our problem, although the number of possible range/point constraints is infinite, but we can assume a cardinality/cost domain, say `[0, 10K]`, for the queries to be generated, given the database **D**. Then we can divide the domain uniformly into K subranges, each of which corresponds to a constraint, *i.e.*, a task, and thereby we have multiple tasks. We use the notation C to denote the set of constraints. Suppose $K = 5$, and the C = {`[0, 2K]`, `[2K, 4K]`, `[4K, 6K]`, `[6K, 8K]`, `[8K, 10K]`}.

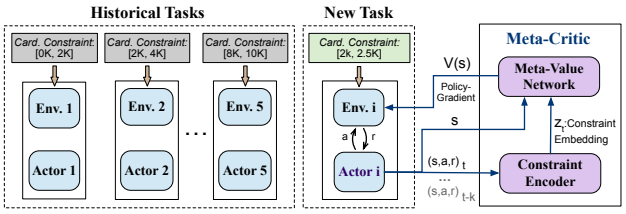


Figure 3: The LearnedSQLGen Architecture.

Next, we can train with these tasks (actors) together to obtain a meta-critic network. When it comes to a new task with a constraint (within the domain `[0, 10K]`), we can leverage the knowledge obtained from the previous K tasks to better train the new task.

Next, we show the concrete mechanisms of the training and inference steps mainly for range constraints. For the point constraint, the method is similar and we will discuss it at the end of this section.

Meta-critic training: In this part, given the database **D** and the constraint set C (produced from the domain), we focus on how to train a meta-critic network and reveal its intrinsic characteristic that can help to generalize to a new task efficiently and effectively. To this end, in the *actor-critic* framework, it is significant for the *critic* network to estimate the long-term reward, *i.e.*, the V value accurately, which guides the actor to generate proper queries. Hence, the goal of the meta-critic network is to make the new task estimate an accurate V value in few trials efficiently.

To achieve this, the meta-critic network incorporates multiple actors, each of which corresponds to a constraint (task) in C. Another key factor is that it adds a constraint encoder, which potentially embeds the information w.r.t. different tasks. Hence, the meta-critic network has captured the relationship between different tasks and their corresponding V values, given some generated queries. Once a new task arrives, the meta-critic can capture the feature of the new task through the constraint encoder, and leverage the learned knowledge of previous tasks that have close relationships with the new one to generalize quickly. Moreover, the meta-critic keeps learning to criticize actors from new tasks, it accumulates transferable knowledge and never gets 'out of date'.

As shown in Figure 3, initially given the domain `[0, 10K]`, we divide it to 5 tasks (actors) with 5 successive range constraints. Then the training process begins. First, these *actors* one-by-one generate SQL queries served as the training data. For each task, the query generation way is the same as discussed in Section 3.3. For example, the first task in Figure 3 aims to generate queries with cardinality in `[0, 2K]`. Second, for the actor-critic network in the previous section, the generated query is sent to the environment for execution, and the returned reward is utilized to update both networks. Different from that, the meta-critic framework adds a *constraint encoder*, which embeds the current state (*i.e.*, the query), selected action (*i.e.*, the token) and the returned reward by the environment, denoted by a triple (s_t, a_t, r_t). Then the outputs of the constraint encoder, denoted by z_t, can potentially describe the task, because the task directly determines the reward, given the query and selected token. Third, the *meta-value* network takes as input (s_t, a_t, z_t), and estimates the long-term reward, *i.e.*, the V value. Different from the critic network in basic actor-critic model, the meta-value network estimates the long-term reward based on both the generated queries and constraints, which considers the historical training experience of different constraints.

Next formally, with the estimated V value, the actor is trained by $\nabla J(\theta) = \mathbb{E}_{\pi_\theta}(\sum_{t=0}^{T} \nabla_\theta log\pi_\theta(a_t|s_t)A(s_t, a_t, z_t))$, where the critic can give relatively accurate estimation of $A(s_t, a_t, z_t)$ (based on the V value) and efficiently guide the actor to find proper queries. Similar to Section 4, we then update both the meta-value network and constraint encoder based on the loss function, $\mathcal{L}_\phi = (r_t + V_\phi(s_{t+1}, z_t) - V_\phi(s_t, z_t))^2$, which approximates the distance from the actual long-term reward under the new constraint.

Meta-critic generalization: As discussed before, given a new task, if the meta-value network can estimate the V value accurately and efficiently, the actor can quickly adjust its policy to generate queries satisfy the new constraint. Recap that the meta-value network in the meta-critic framework can capture the feature of different tasks (incorporated by the constraint encoder), the queries(*i.e.*, state) and output the V value. Hence, for a new task, leveraging the learned meta-critic framework, one can identify the new task's feature and use the previously learned knowledge to quickly estimate an accurate V value, which leads to efficient actor training. For example, given a new task of generating SQL queries satisfying a constraint [1k, 2.5k], we can mostly leverage the learned knowledge from pre-trained similar tasks, *i.e.*, the constraints $[0k, 2k]$ and $[2k, 4k]$, to quickly train the new task. Furthermore, after the new task is trained, the knowledge of the task is also incorporated into the meta-critic network, and thus the network will become more and more powerful.

Remark for the point constraint. We can sample some points in the domain to pre-train the meta-critic network. For example, we can sample a set of cardinality points $C = \{10^1, 10^2, 10^3, 10^4, 10^5\}$ with different magnitude and train a meta-critic network. Then, given a new task of `Card` = 2K, the meta-critic efficiently learn the task potentially leveraging the knowledge of the previous tasks.

7 EXPERIMENTS

We have conducted extensive experiments to show the superiority of our proposed `LearnedSQLGen` framework.

7.1 Experiment Setting

Datasets. We used three benchmark datasets which are widely used in our database community. (1) `TPC-H` [2] is a popular benchmark that contains 8 relational tables (We produce the data with size of 33 GB in total). Given these tables, `LearnedSQLGen` can generate SQL queries that can be executed over them. (2) `JOB` [1] is another widely-used benchmark (uses a real-world dataset IMDB) that consists of 21 tables (with size of 14 GB in total). (3) `XueTang` [3] is a real-world OLTP benchmark of the online education, which contains 14 tables (with size of 24 GB in total).

Baselines. We compared with two baselines in the paper.
(1) `SQLSmith` [39] randomly generated SQLs based on a parse tree, from which we picked the queries satisfied the constraints.
(2) `Template` [10] generated satisfied SQL queries by greedily tweaking the predicate values in the given SQL templates. The query templates are constructed from the provided templates of the three benchmarks by reassembling the predicates (e.g., adding or removing a predicate).

Hyper-parameter Settings. The actor network consists of an input layer, followed by a 2-layer LSTM network and an output layer.

Note that the number of layers is chosen by balancing efficiency and accuracy. A more complex model is likely to achieve better results but may sacrifice efficiency, and thus we can take it as a hyper-parameter tuning problem. In practice, we find that 2-layer is most appropriate. The dimension of the input/output layer is equal to the size of the action space of each dataset. Here, we have the dimension 1962 (TPC-H), 3940 (JOB) and 4280 (XueTang) respectively. We set the sampled numbers of values in each numerical attribute as 100, *i.e.*, $k = 100$. The 2-layer LSTM networks have 20 cell units respectively and the second one is connected to the output layer. The structure of the critic network is similar to the actor, but the only difference is that the output layer dimension is 1 for outputting the V value. Dropout [42] is used to avoid over-fitting in both the actor and critic networks and it is set to 0.3. We set $\lambda = 0.01$ to prevent the actor from generating a lot of same queries. In the training process, the learning rate is 0.001 for the actor and 0.003 for the critic network.

Target Constraints. We evaluated two types of target constraints. (1) `Cost` denotes the execution expense of an SQL and (2) `Cardinality` denotes the size of the result of an SQL. We directly used the estimated cardinality/cost by the database estimator to compute the reward. Furthermore, we respectively tested the two types of constraints in forms of points and ranges.

Evaluation Metrics. We evaluated `LearnedSQLGen` using two metrics, *i.e.*, generation accuracy and generation time.

(1) Generation accuracy (*acc*) represents the ratio of the number of satisfied queries (denoted by n_s) to the total number of generated queries (denoted by n). Then $acc = \frac{n_s}{n}$. A higher accuracy reflects that the method is more powerful in generating satisfied SQLs. For the point constraint, it is hard for the cardinality/cost c to be a value exactly, so a small error bound τ is set, which means that if the cardinality/cost of a generated query is within $[c - \tau, c + \tau]$, we regard it as a satisfied query. To be specific, we set τ as $0.1 * c$, which is reasonable because a 10% deviation from c is acceptable. Hence, n_s denotes the number of such queries. For the range constraint, n_s just denotes the number of queries in the range.

(2) Generation time represents the time of generating a fixed number (*e.g.*, 1K) of satisfied SQLs including the training and inference phases. A lower generation time indicates that the generation method is more efficient.

Environment. All experiments were implemented in Python, performed on a Ubuntu Server with an Intel (R) Xeon (R) Silver 4110 2.10GHz CPU having 32 cores, a Nvidia Geforce 2080ti GPU, and 128GB DDR4 main memory without SSD.

7.2 Overall Evaluation

In this subsection, we compared the accuracy and efficiency of `LearnedSQLGen` with the random-based method (`SQLSmith`) and template-based heuristic method (`Template`).

7.2.1 Accuracy of generated queries. We first evaluated the accuracy for cardinality constraint and then the cost.

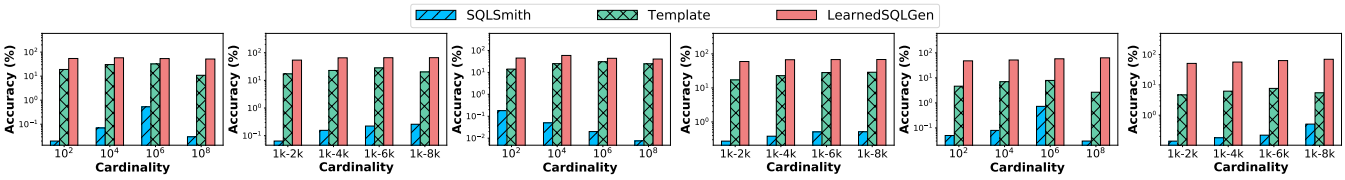
Varying the Cardinality. As shown in Figure 4, we generated 1K queries on three datasets, and tested the accuracy of different point and range cardinality constraints and the results showed that `LearnedSQLGen` outperformed the baselines. For example, in the x-axis of Figure 4(a), `Cardinality` $= 10^2$ denotes that we aim to generate queries with cardinality as close to 10^2 as possible, *i.e.*, queries with cardinality between 90 and 110 are satisfied ones. On `TPC-H`, `LearnedSQLGen` achieved an accuracy of 54.33%, while `SQLSmith` and `Template` had 0.02% and 18.98% respectively. Thus, `LearnedSQLGen` outperformed `SQLSmith` because `SQLSmith` generated queries randomly without considering the constraints, but `LearnedSQLGen` incorporated the constraints and the execution feedback from the database into reward computation in the design of our reinforcement learning model. Hence, we can capture the relationship between the SQL query and its cardinality/-cost, and thus the generation direction is guided by the model. In Figure 4(a), `SQLSmith` gained the highest accuracy (*i.e.*, 0.53%) at `Cardinality` $= 10^6$ mainly because, on this dataset, the cardinality of random generated queries is close to 10^6 than other point constraints, but it is still much lower than that of `LearnedSQLGen`. `LearnedSQLGen` outperformed `Template` because the templates in `Template` were not guaranteed to be high-quality (*i.e.*, can be adjusted to meet the given constraint), and thus some of them cannot produce the satisfied queries. As shown in Figures 4(c,e), `LearnedSQLGen` still outperformed the two baselines. For example, in Figure 4(c), when the cardinality was `Cardinality` $= 10^4$, `LearnedSQLGen` achieved an accuracy of 59.30%, while `SQLSmith` and `Template` were 0.05% and 24.94% respectively.

We also tested the accuracy for different range constraints. For example, as shown in Figure 4(b), `Cardinality` $\in$ `[1K, 2K]` denotes that we aim to generate satisfied queries with cardinality within this range. On `TPC-H`, `LearnedSQLGen` still outperformed the baselines for similar reasons. `LearnedSQLGen` achieved an accuracy of 54.12%, while `SQLSmith` and `Template` had 0.064% and 17.16% respectively. We can see that as the range gets wider, the accuracy of all the three methods is generally higher. For example, for `LearnedSQLGen`, compared with `Cardinality` $\in$ `[1K, 2K]` (54.12%), the accuracy is 66.23% when `Cardinality` $\in$ `[1K, 8K]`. This is because there will be naturally more queries satisfying a wider range constraint.

Varying the Cost. Our `LearnedSQLGen` framework can also support the cost constraint, as shown in Figure 5. Due to the similar reasons, `LearnedSQLGen` still outperforms the baselines. For example, for the point constraint on `XueTang` dataset, when `Cost` $= 10^6$, `LearnedSQLGen` achieves an accuracy of 53.66%, which is much better than that of `SQLSmith` (0.24%) and `Template` (18.11%). For the range constraint on `JOB` dataset, `Cost` $\in$ `[1K, 8K]`, `LearnedSQLGen` achieves an accuracy of 51.37%, while `SQLSmith` and `Template` are 2.17% and 20.15% respectively.

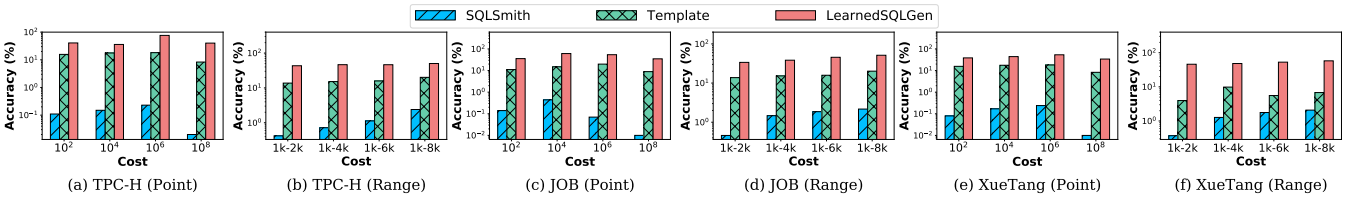
7.2.2 Efficiency of the query generation. We tested the efficiency of `LearnedSQLGen` to generate satisfied queries.

Varying the Cardinality. In Figure 6, we evaluated the efficiency of different cardinality constraints on three datasets. Overall, we can see that our method outperformed all the baselines. For example, as shown in Figure 6(a), on `TPC-H` dataset, if we want to generate queries close to 10^8, `LearnedSQLGen` only consumed 0.63 hours, but `SQLSmith` and `Template` had to spend 11 hours and 2.72 hours respectively. The reason is that `SQLSmith` just randomly generated queries, and thus it took a longer time to generate 1K satisfied queries. `Template` took a longer time because some templates cannot produce satisfied queries by just exploring operands



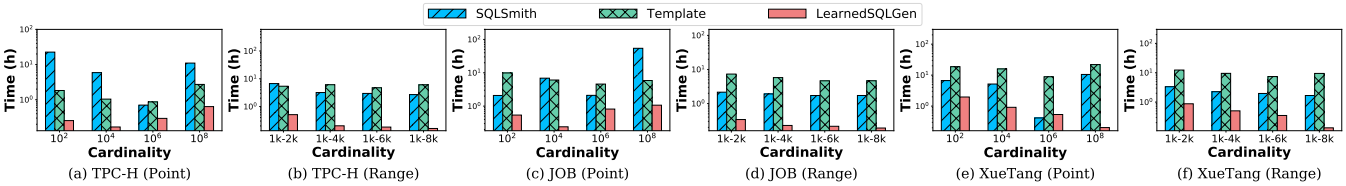
(a) TPC-H (Point) (b) TPC-H (Range) (c) JOB (Point) (d) JOB (Range) (e) XueTang (Point) (e) XueTang (Range)

Figure 4: Accuracy Evaluation of Cardinality Constraint (N=1000).



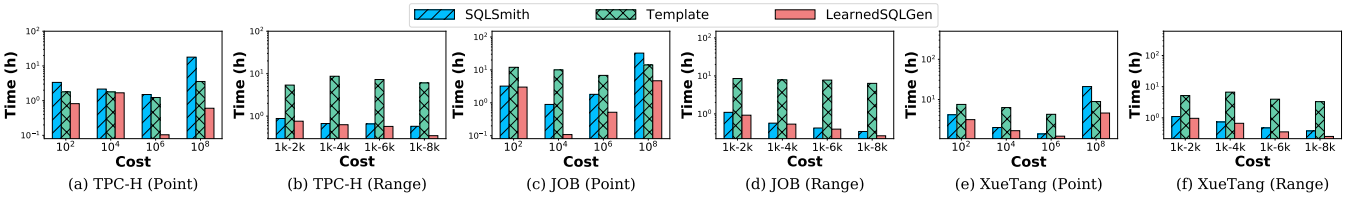
(a) TPC-H (Point) (b) TPC-H (Range) (c) JOB (Point) (d) JOB (Range) (e) XueTang (Point) (f) XueTang (Range)

Figure 5: Accuracy Evaluation of Cost Constraint (N=1000).



(a) TPC-H (Point) (b) TPC-H (Range) (c) JOB (Point) (d) JOB (Range) (e) XueTang (Point) (f) XueTang (Range)

Figure 6: Efficiency Evaluation of Cardinality Constraint (N=1000).



(a) TPC-H (Point) (b) TPC-H (Range) (c) JOB (Point) (d) JOB (Range) (e) XueTang (Point) (f) XueTang (Range)

Figure 7: Efficiency Evaluation of Cost Constraint (N=1000).

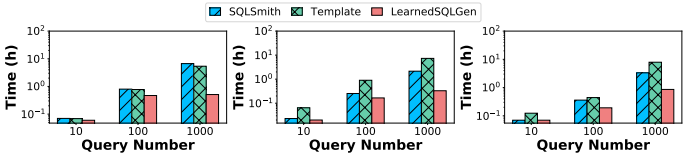
in predicates but keeping template structures fixed. This wasted a lot of time on the given templates, so it performed even worse than SQLSmith. For example, when Template aimed to generate queries satisfying the constraint Cardinality = 10^8 on TPC-H, Template took as input a template Select * From Customer Where Customer.c_custkey < x but it can never reach 10^8 by adjust parameter x because the total number of rows in the table Customer is less than 10^8. From this perspective, LearnedSQLGen is more effective as it explores both the query structures (*i.e.*, templates) and the parameters. For range constraints, we have similar observations. For example, as shown in Figure 6(d), on JOB dataset, if we want to generate queries with cardinality within Cardinality ∈ [1K, 6K] , LearnedSQLGen only consumed 0.21 hours, while SQLSmith and Template had to spend 1.71 hours and 4.643 hours respectively.

Varying Cost. We also tested the efficiency of different methods with respect to the cost constraint. As shown in Figure 7. We can observe that LearnedSQLGen outperformed the baselines. For example, for the point constraint on TPC-H dataset, when Cost = 10^2, LearnedSQLGen spent 0.8 hours, which is much more efficient than that of SQLSmith (3.33 hours) and Template (1.78 hours). For the range constraint Cost ∈ [1K, 2K] on XueTang dataset, LearnedSQLGen took 0.96 hours, while SQLSmith and Template consumed 1.1 hours and 5.18 hours respectively.

Varying the number of satisfied queries. We varied the number of satisfied queries to be generated, *i.e.*, 10, 100, 1000 respectively. As shown in Figures 8-9, we can observe that, for generating different numbers of satisfied queries (represented on the X-axis), LearnedSQLGen can be faster than baselines on three dataset. For example, for the cardinality constraint Cardinality ∈ [1K, 2K] on JOB, shown in Figure 8(b), we aim to generate 100 queries, LearnedSQLGen spent 0.166 hours while SQLSmith and Template took 0.25 hours and 0.9 hours respectively. For the cost constraint Cost ∈ [1K, 2K] on XueTang, shown in Figure 9(c), when generate 1k satisfied queries, LearnedSQLGen took 0.9 hours while SQLSmith and Template consumed 1.1 and 9.5 hours respectively.

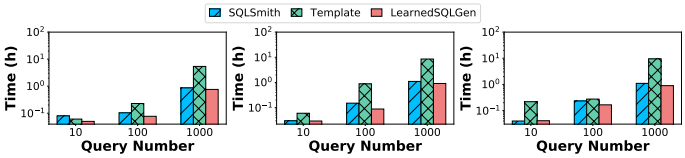
7.3 Evaluation of the actor-critic network

We evaluated the efficiency of the actor-critic (AC) algorithm in LearnedSQLGen by comparing with the simple reinforcement learning algorithm (REINFORCE) mentioned in Section 4.3 on solving constraint-aware query generation problem. We had three observations. First, we can observe that, by leveraging the actor-critic network, we can perform better than using the REINFORCE algorithm on accuracy. Specifically, as shown in Figure 10(a), on JOB dataset, given the constraint Cardinality ∈ [1K, 4K], LearnedSQLGen achieved an accuracy of 65.43%, which is 9.2% higher than that of REINFORCE. The reason is that our method can reduce the variance



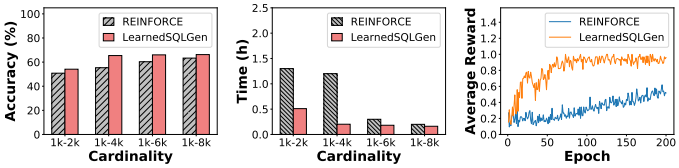
(a) TPC-H (b) JOB (c) XueTang

Figure 8: Efficiency Evaluation by Generating Different Numbers of Satisfied Queries (Cardinality in [1k, 2k]).



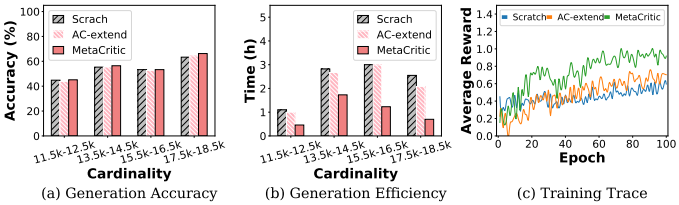
(a) TPC-H (b) JOB (c) XueTang

Figure 9: Efficiency Evaluation by Generating Different Numbers of Satisfied Queries (Cost in [1k, 2k]).



(a) Generation Accuracy (b) Generation Efficiency (c) Training Trace

Figure 10: Comparison of RL Algorithms (TPC-H, N=1000).



(a) Generation Accuracy (b) Generation Efficiency (c) Training Trace

Figure 11: Comparison of Meta-critic (TPC-H, N=1000).

of the returned rewards and keeps policy gradient update steadily, which, to some extent, prevents the network from converging to the local optimum. Second, as shown in Figure 10(b), our method is more efficient than the REINFORCE algorithm. For example, when `Cardinality` $\in$ `[1K, 4K]`, `LearnedSQLGen` spent 0.51 hours to acquire 1K satisfied queries while REINFORCE took 2.2 hours. The reason is that `LearnedSQLGen` converged faster and was more accurate than REINFORCE, such that it took less time to acquire the same number of satisfied queries. Third, in Figure 10(c), we showed the training process of REINFORCE and `LearnedSQLGen`. It illustrated that with the number of training epochs increasing, the returned reward of `LearnedSQLGen` was much higher than that of REINFORCE. This indicated that our method converged more steadily and achieved a higher performance.

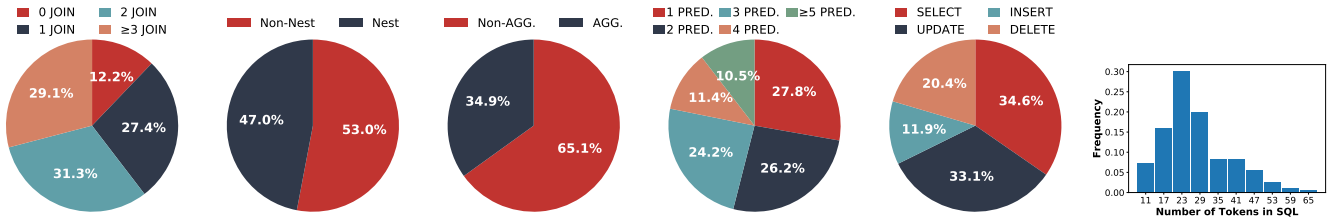
7.4 Meta-critic Network Evaluation

We evaluated our meta-critic network that can generalize our model to different constraints. We compared with three strategies. (1) Given a new constraint, we trained `LearnedSQLGen` from scratch (`Scrach`); (2) We used the pre-trained meta-critic network to quickly generalize to the new constraint (`MetaCritic`). (3) We directly encoded multiple constraints to the state without using the meta-critic, `AC-extend`. Specifically, we tested the point constraint for cardinality. On `XueTang` dataset, we set the domain as [10K, 20K], which was divided into 10 parts, C = {`[10, 11K]`, `[11K, 12K]`, ..., `[19K, 20K]`}, and then trained the meta-critic network. Next, given new constraints within this domain, we leveraged previous learned knowledge to efficiently train the new tasks. We respectively tested the new constraints [11.5K, 12.5K], [13.5K, 14.5K], [15.5K, 16.5K], [17.5K, 18.5K], and the results are shown in Figure 11. For the `AC-extend` baseline, we also set the domain as [10K, 20K], divided it into 10 parts (*i.e.*, tasks) and generated queries for each task. The only difference was that we encoded the constraint (e.g., $[10K, 11K]$) into the state and then trained the actor-critic network. Thus the agent captures the relationship between the constraint and reward under a certain state.

Overall, we made three observations. First, `MetaCritic` significantly reduced the query generation time, because `MetaCritic` learned from historical training experience, provided relatively accurate long-term reward, and efficiently guided the agent to select proper tokens to generate SQLs. Instead, `Scrach` needed to learn the estimation of long-term reward from scratch, and thus took much longer time. Compared with `AC-extend`, our meta-critic method also encoded the constraints, but in a more fine-grained way. More concretely, we potentially specified the constraints based on the triple (state, action, reward) because the state and action can specify a query (subquery), and its reward had a close relation with the constraint. Then, using the triple as the constraint encoder captured more transferrable knowledge among different tasks, and thus leaded to good generalization ability. However, `AC-extend` cannot generalize well because it is hard to capture the relationships among tasks by purely encoding the tasks. Second, as shown in Figure 11(a), all the three baselines can achieve high accuracy, because they can finally converge to a relatively optimal generation policy. However, `MetaCritic` had a slightly higher accuracy than `Scrach` and `AC-extend` because it considered both the historical experience and the new constraint, which was captured automatically by the triple (state, action, reward). But `Scrach` purely learned from newly generated query samples and `AC-extend` captured the constraint by simply feeding into the neural network. Third, as shown in Figure 11(c), although these baselines had similar performance at initial epochs, as the number of training epochs grew, `MetaCritic` quickly generated satisfied queries guided by more accurate long-term rewards, while `Scrach` and `AC-extend` took more time to estimate the long-term rewards.

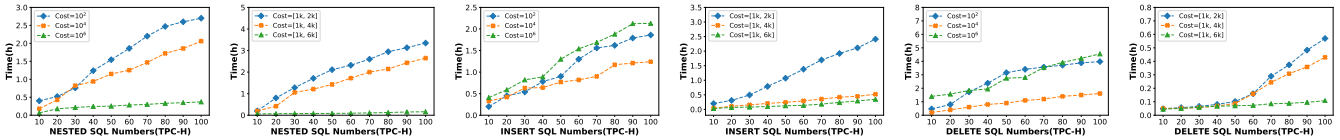
7.5 Case Study of Generated Queries

We evaluate the diversity and complexity of the generated queries, and report the query distributions from different perspectives, including predicate numbers, aggregation keywords, nested queries, join queries, SQL types, and number of SQL tokens. In Figure 12(a), the satisfied queries are likely associated with multi-join tables. We can see the ratio of queries with multi-join is over half of total generated queries. In Figure 12(b) and (c), we can see that many complicated structures, *e.g.*, nested (47%) and aggregation (34.9%) queries, can be generated. In Figure 12(d)-(e), we can generate diverse queries. For the low cardinality range in [1k, 8k], the satisfied



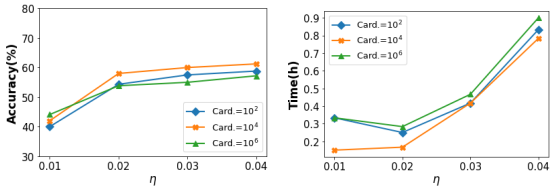
(a) Join Tables (b) Nested Query (c) Aggregates (d) Query Predicates (e) Query Types (f) SQL Lengths

Figure 12: Evaluation of Generated Query Distribution. (a), (b), (c) and (f): Generating 1K Queries with `Cost` = 10^6 on `TPC-H`; (d) and (e): Generating 1K Queries with `Cardinality` in [1k, 8k] on `TPC-H`.



(a) Nested (Point) (b) Nested (Range) (c) Insert (Point) (d) Insert (Range) (e) Delete (Point) (f) Delete (Range)

Figure 13: Efficiency Evaluation of Generating Different Numbers of Complicated Queries with Cost Constraint (TPC-H).



(a) TPC-H (Point) (b) TPC-H (Point)

Figure 14: Evaluation on Sample Sizes (TPC-H, N=1000).

queries usually contain multiple predicates (*e.g.*,, many "and") to reduce the cardinality. Hence, in Figure 12(d), we can see that there are various number of predicates in the generated queries. Also, because of the entropy regularization techniques and our extendable FSM, our method can support various query types as shown in Figure 12(e). At last, Figure 12(f) shows the SQL length distribution, where the x-axis denotes the number of tokens and the y-axis denotes the frequency. We can observe that `LearnedSQLGen` can generate various lengths of queries, which verifies that we can generate diverse and complicated queries.

7.6 Complicated Queries Generation

We have added experiments to evaluate the performance of generating complicated queries in Figure 13, where the x-axis denotes the number of different types of generated complex queries (including nested, insert, delete queries), and the y-axis denotes the time spent to generate these nested queries that satisfy the constraint (specified by the legend, *e.g.*, `Cost`=10^2). From the results, we can observe that `LearnedSQLGen` can generate various types of queries by extending the FSM. Thus our method is applicable to generate various of complicated SQL queries.

7.7 Evaluation on Sample Sizes

We evaluate the impact of different sampled sizes of numerical values in Figure 14. The x-axis is the sample ratio η which is the ratio of the sample size to the total number of distinct values in a column. We evaluate both the point and range constraints in this experiment. For accuracy, we can see from Figure 14(a) that with the sample size becoming larger, the accuracy becomes higher because more actions can be chosen. Moreover, the accuracy keeps stable after a number of data have been sampled, because a certain number of samples can cover a very large search space and it is enough to cover the queries satisfying the user-specified constraints. Therefore, our framework is not much sensitive to the sample size. For efficiency, we can observe that with η becoming larger, the spent time decreases at the beginning and then increases. This is because SQL generation time includes training time and inference time, and increasing the sample size makes training slow (a larger search space) but accelerates inference (higher SQL coverage). Thus at the beginning, it slightly increases the training time but significantly decreases the inference time, and thus the total time is smaller; but later, it significantly increases the training time, and thus the total time becomes larger.

8 CONCLUSION

In this paper, we have studied SQL queries generation by considering the target constraints including cardinality and cost. We designed an RL framework to conduct query generation, where an exploration-exploitation strategy was applied to exploit the optimal generation direction and explore multiple possible directions. In addition, we adopted an FSM to guarantee that we can generate valid queries. Moreover, to improve generalization ability, we proposed a meta-critic network that learned from historic and used the model to directly generate queries for a new constraint. Experimental results showed that our method significantly outperformed baselines in terms of both accuracy and efficiency.

ACKNOWLEDGEMENT

This paper was supported by NSF of China (61925205, 61632016, 62102215), Huawei, TAL education, and Beijing National Research Center for Information Science and Technology (BNRist). Chengliang Chai is supported by China National Postdoctoral Program for Innovative Talents (BX2021155), Postdoctoral Foundation(2021M691784), and Zhejiang Lab's International Fund for Young Professionals.

REFERENCES

[1] Job benchmark. https://github.com/gregrahn/join-order-benchmark.
[2] Tpch benchmark. http://www.tpc.org.
[3] Xuetang dataset. https://www.xuetangx.com/.
[4] R. Ahmed, R. G. Bello, A. Witkowski, and P. Kumar. Automated generation of materialized views in oracle. *Proc. VLDB Endow.*, 13(12):3046–3058, 2020.
[5] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. Learning-based query performance modeling and prediction. In *ICDE*, pages 390–401, 2012.
[6] D. V. Aken, D. Yang, S. Brillard, A. Fiorino, B. Zhang, C. Billian, and A. Pavlo. An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. *Proc. VLDB Endow.*, 14(7):1241–1253, 2021.
[7] D. Bahdanau, P. Brakel, K. Xu, A. Goyal, R. Lowe, J. Pineau, A. C. Courville, and Y. Bengio. An actor-critic algorithm for sequence prediction. In *ICLR*. OpenReview.net, 2017.
[8] H. Bati, L. Giakoumakis, S. Herbert, and A. Surna. A genetic approach for random testing of database systems. In *VLDB*, VLDB '07, page 1243–1251. VLDB Endowment, 2007.
[9] L. Battle, P. Eichmann, M. Angelini, T. Catarci, G. Santucci, Y. Zheng, C. Binnig, J. Fekete, and D. Moritz. Database benchmarking for supporting real-time interactive querying of large data. In *SIGMOD*, pages 1571–1587, 2020.
[10] N. Bruno, S. Chaudhuri, and D. Thomas. Generating queries with cardinality constraints for dbms testing. volume 18, pages 1721–1725, 2006.
[11] V. Christophides, V. Efthymiou, T. Palpanas, G. Papadakis, and K. Stefanidis. End-to-end entity resolution for big data: A survey. *CoRR*, abs/1905.06397, 2019.
[12] B. Ding, S. Chaudhuri, J. Gehrke, and V. R. Narasayya. DSB: A decision support benchmark for workload-driven and traditional database systems. *Proc. VLDB Endow.*, 14(13):3376–3388, 2021.
[13] J. Ding, U. F. Minhas, J. Yu, and et al. ALEX: an updatable adaptive learned index. In *SIGMOD*, pages 969–984. ACM, 2020.
[14] P. Dintyala, A. Narechania, and J. Arulraj. Sqlcheck: Automated detection and diagnosis of SQL anti-patterns. In *SIGMOD*, pages 2331–2345, 2020.
[15] A. Gruenheid, S. Deep, K. Nagaraj, H. Naito, J. F. Naughton, and S. Viglas. Diametrics: Benchmarking query engines at scale. *Proc. VLDB Endow.*, 13(12):3285–3298, 2020.
[16] Y. Han, Z. Wu, P. Wu, R. Zhu, J. Yang, L. W. Tan, K. Zeng, G. Cong, Y. Qin, A. Pfadler, Z. Qian, J. Zhou, J. Li, and B. Cui. Cardinality estimation in dbms: A comprehensive benchmark evaluation, 2022.
[17] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.
[18] S. Idreos, N. Dayan, W. Qin, M. Akmanalp, S. Hilgard, A. Ross, J. Lennon, V. Jain, H. Gupta, D. Li, and Z. Zhu. Design continuums and the path toward self-designing key-value stores that know and learn. In *CIDR*, 2019.
[19] X. Kang, Y. Zhao, J. Zhang, and C. Zong. Dynamic context selection for document-level neural machine translation via reinforcement learning. In B. Webber, T. Cohn, Y. He, and Y. Liu, editors, *EMNLP*, pages 2242–2254. Association for Computational Linguistics, 2020.
[20] G. Li, X. Zhou, and L. Cao. AI meets database: AI4DB and DB4AI. In *SIGMOD*, pages 2859–2866, 2021.
[21] G. Li, X. Zhou, and L. Cao. Machine learning for databases. *Proc. VLDB Endow.*, 14(12):3190–3193, 2021.
[22] G. Li, X. Zhou, S. Li, and B. Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *PVLDB*, 12(12):2118–2130, 2019.
[23] J. Li, A. C. Konig, V. Narasayya, and S. Chaudhuri. Robust estimation of resource consumption for sql queries using statistical techniques. *PVLDB*, 5(11):1555–1566, 2012.
[24] Z. Li, J. Kiseleva, and M. de Rijke. Rethinking supervised learning and reinforcement learning in task-oriented dialogue systems. In T. Cohn, Y. He, and Y. Liu, editors, *EMNLP*, volume EMNLP 2020 of *Findings of ACL*, pages 3537–3546. Association for Computational Linguistics, 2020.
[25] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. In Y. Bengio and Y. LeCun, editors, *ICLR*, 2016.
[26] X. Liu, X. Kong, L. Liu, and K. Chiang. In *ICDM*, pages 1140–1145, 2018.
[27] L. Ma, W. Zhang, J. Jiao, W. Wang, M. Butrovich, W. S. Lim, P. Menon, and A. Pavlo. MB2: decomposed behavior modeling for self-driving database management systems. In *SIGMOD*, pages 1248–1261, 2021.
[28] 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.
[29] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. Bao: Learning to steer query optimizers. *CoRR*, abs/2004.03814, 2020.
[30] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. In R. Bordawekar and O. Shmueli, editors, *aiDM@SIGMOD*, pages 3:1–3:4. ACM, 2018.
[31] C. Mishra, N. Koudas, and C. Zuzarte. Generating targeted queries for database testing. In *SIGMOD*, 2008.
[32] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In M. F. Balcan and K. Q. Weinberger, editors, *ICML*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR.
[33] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra. Deep learning for entity matching: A design space exploration. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *SIGMOD*, pages 19–34, 2018.
[34] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning multi-dimensional indexes. In *SIGMOD*, pages 985–1000, 2020.
[35] K. Nguyen, H. D. III, and J. L. Boyd-Graber. Reinforcement learning for bandit neural machine translation with simulated human feedback. In M. Palmer, R. Hwa, and S. Riedel, editors, *EMNLP*, pages 1464–1474. Association for Computational Linguistics, 2017.
[36] M. Ranzato, S. Chopra, M. Auli, and W. Zaremba. Sequence level training with recurrent neural networks. In Y. Bengio and Y. LeCun, editors, *ICLR*, 2016.
[37] Y. Remil, A. Bendimerad, R. Mathonat, P. Chaleat, and M. Kaytoue. What makes my queries slow?: Subgroup discovery for SQL workload analysis. In *ASE*, pages 642–652, 2021.
[38] F. Savva, C. Anagnostopoulos, and P. Triantafillou. ML-AQP: query-driven approximate query processing based on machine learning. *CoRR*, abs/2003.06613, 2020.
[39] A. Seltenreich. Sqlsmith, 2020. https://github.com/anse1/sqlsmith.
[40] S. P. Singh, M. J. Kearns, D. J. Litman, and M. A. Walker. Reinforcement learning for spoken dialogue systems. In S. A. Solla, T. K. Leen, and K. Müller, editors, *NIPS*, pages 956–962. The MIT Press, 1999.
[41] D. Slutz. Massive stochastic testing of sql. In *VLDB*, 1998.
[42] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.
[43] F. Sung, L. Zhang, T. Xiang, T. M. Hospedales, and Y. Yang. Learning to learn: Meta-critic networks for sample efficient learning. *CoRR*, abs/1706.09529, 2017.
[44] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
[45] J. Wang, C. Chai, J. Liu, and G. Li. FACE: A normalizing flow based cardinality estimator. *Proc. VLDB Endow.*, 15(1):72–84, 2021.
[46] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, 1992.
[47] R. J. Williams and J. Peng. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3(3):241–268, 1991.
[48] L. Wu, F. Tian, T. Qin, J. Lai, and T. Liu. A study of reinforcement learning for neural machine translation. In E. Riloff, D. Chiang, J. Hockenmaier, and J. Tsujii, editors, *EMNLP*, pages 3612–3621. Association for Computational Linguistics, 2018.
[49] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *ICDE*, pages 1081–1092, 2013.
[50] X. Yu, G. Li, C. Chai, and N. Tang. Reinforcement learning with tree-lstm for join order selection. In *ICDE*, pages 1297–1308. IEEE, 2020.
[51] H. Yuan, G. Li, L. Feng, and et al. Automatic view generation with deep learning and reinforcement learning. In *ICDE*, pages 1501–1512, 2020.
[52] J. Zhang, Y. Liu, K. Zhou, and G. Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *SIGMOD*, pages 415–432, 2019.
[53] R. Zhong, Y. Chen, H. Hu, H. Zhang, W. Lee, and D. Wu. SQUIRREL: testing database management systems with language validity and coverage feedback. In *CCS*, pages 955–970, 2020.
[54] X. Zhou, C. Chai, G. Li, and J. Sun. Database meets artificial intelligence: A survey. *TKDE*, 2020.
[55] X. Zhou, L. Jin, S. Ji, and et al. Dbmind: A self-driving platform in opengauss. *Proc. VLDB Endow.*, 14(12):2743–2746, 2021.
[56] X. Zhou, G. Li, C. Chai, and J. Feng. A learned query rewrite system using monte carlo tree search. *PVLDB*, 2022.
[57] X. Zhou, L. Liu, W. Li, and et al. Autoindex: An incremental index management system for dynamic workloads. In *ICDE*, 2022.
[58] X. Zhou, J. Sun, G. Li, and J. Feng. Query performance prediction for concurrent queries using graph embedding. *Proc. VLDB Endow.*, 13(9):1416–1428, 2020.