# ExeSQL: Self-Taught Text-to-SQL Models with Execution-Driven Bootstrapping for SQL Dialects

**Jipeng Zhang**[1*], **Haolin Yang**[1*], **Kehao Miao**[2*], **Ruiyuan Zhang**[1], **Renjie Pi**[1], **Jiahui Gao**[3], **Xiaofang Zhou**[1]
[1]The Hong Kong University of Science and Technology
[2]Nanyang Technological University, [3]The University of Hong Kong
{jzhanggr,hyangby,zry,rpi,zxf}@ust.hk
kehao001@e.ntu.edu.sg, ggaojiahui@gmail.com

## Abstract

Recent text-to-SQL models have achieved strong performance, but their effectiveness remains largely confined to SQLite due to dataset limitations. However, real-world applications require SQL generation across multiple dialects with varying syntax and specialized features, which remains a challenge for current models.

The main obstacle in building a dialect-aware model lies in acquiring high-quality dialect-specific data. Data generated purely through static prompting—without validating SQLs via execution—tends to be noisy and unreliable.Moreover, the lack of real execution environments in the training loop prevents models from grounding their predictions in executable semantics, limiting generalization despite surface-level improvements from data filtering. This work introduces ExeSQL, a text-to-SQL framework with execution-driven, agentic bootstrapping. The method consists of iterative query generation, execution-based filtering (e.g., rejection sampling), and preference-based training, enabling the model to adapt to new SQL dialects through verifiable, feedback-guided learning. Experiments show that ExeSQL bridges the dialect gap in text-to-SQL, achieving average improvements of 15.2%, 10.38%, and 4.49% over GPT-4o on PostgreSQL, MySQL, and Oracle, respectively, across multiple datasets of varying difficulty.

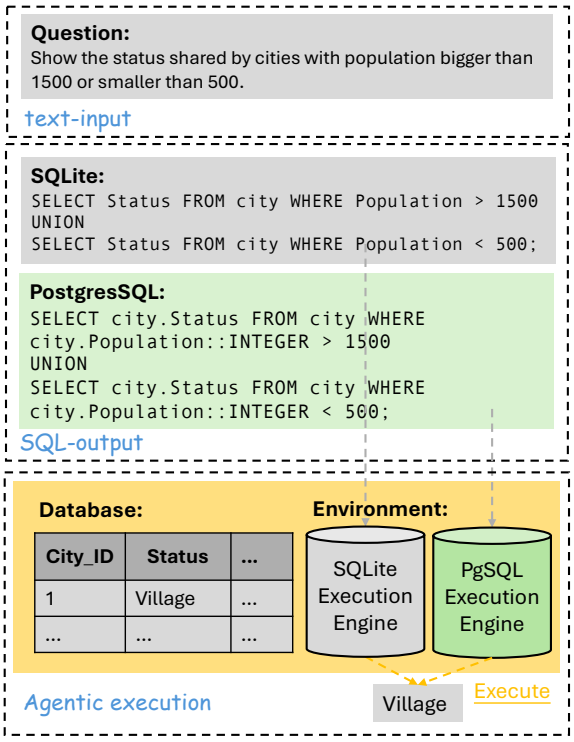


Figure 1: Given a natural language question, different SQL dialects require distinct syntax adjustments, such as explicit type casting in PostgreSQL. Beyond the traditional text-input–SQL-output formulation, we incorporate the database environment to enable agentic execution feedback for data synthesis and training.

## 1 Introduction

With the rapid advancement of Large Language Models (LLMs), their capabilities in assisting with various coding tasks have significantly improved. Tools like GitHub Copilot (Microsoft, 2023; Services, 2023) and models such as OpenAI Codex (Chen et al., 2021b) have enhanced developer productivity by automating repetitive tasks, providing real-time suggestions, and offering detailed explanations of code functionality. One crucial application of LLMs in software development is the automatic generation of SQL queries from text (text-to-SQL), a task that has gained increasing attention (Zhong et al., 2017; Yu et al., 2018; Li et al., 2024b; Lei et al., 2024). However, most existing research (Li et al., 2024a; Zhuang et al., 2024; Dong et al., 2023b; Pourreza and Rafiei, 2024; Wang et al., 2023a; Gan et al., 2021; Deng et al., 2021) and datasets in the text-to-SQL domain are primarily designed for SQLite, with limited coverage of widely used database systems such as MySQL, PostgreSQL, BigQuery, Oracle, and

*Equal Contribution. Code are available at the following links: https://github.com/2003pro/exesql.



*Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 24305–24326
November 4-9, 2025 

DuckDB. We incorporate an example of a question with dialect SQL in Figure 1. The lack of high-quality, dialect-specific text-to-SQL data presents significant challenges in developing models that can generalize across different SQL dialects, ultimately hindering the creation of robust and adaptable text-to-SQL solutions for real-world applications (Lei et al., 2024; Li et al., 2024b; Pourreza et al., 2024).

**Rule-Based Translation is Insufficient.** Rule-based translation offers a deterministic but rigid solution to SQL dialect conversion. While transpilers like SQLGlot (Mao, 2023) provide structured mappings between dialects, they struggle with complex syntax, schema constraints, and dialect-specific functions (Zmigrod et al., 2024). Moreover, these systems lack generalizability, require dialect-specific rules (Li et al., 2024b; Lei et al., 2024), and cannot guarantee accurate translation. In practice, they still rely on execution-time feedback to detect and fix failures. Maintaining such rule sets is costly and brittle. Even with carefully crafted rules, such systems cannot guarantee perfect accuracy—particularly for complex or edge-case queries—and often rely on execution-time feedback for correction. We provide a detailed analysis in the Appendix A.10.

**Existing Data Collection and Training Lacks Execution Verification.** General LLM-based code data generation methods (Wei et al., 2023; Wang et al., 2022) often fail to account for the specific requirements of text-to-SQL tasks, leading to the creation of syntactically plausible but incorrect SQL queries. These approaches typically generate large amounts of unverified data, which hinders their usefulness for training reliable models. Since SQL outputs can be directly validated through execution, a more structured approach that incorporates execution-based verification and targeted rejection sampling strategies is necessary. Besides, we argue that standard supervised fine-tuning (SFT) alone is insufficient to fully exploit the potential of execution validation, as it does not inherently enforce correctness across dialects.

To advance dialect text-to-SQL, we emphasize the importance of both high-quality, executable (text, SQL) data and a training pipeline that directly interacts with the execution environment. We propose an agentic data generation loop that combines LLM-based generation, execution-time validation, and self-correction. This offline loop yields reliable training signals, which are distilled into a dialect-aware model through supervised fine-tuning and offline reinforcement learning. The overall workflow includes:

(a) SFT Data Bootstrapping via LLM-based Translation: To mitigate the sparsity of dialect text-to-SQL data and enable effective cold-start training, we leverage high-resource SQLite (text, SQL) pairs and LLMs to efficiently sample dialect SQL queries. This bootstrapped dataset serves as a cold-start fine-tuning set, enabling rapid adaptation to low-resource dialects while minimizing manual annotation.

(b) Iterative SFT Data Generation via Execution-based Rejection Sampling: We extend the dataset via an iterative generation–execution–filtering loop, where the model proposes dialect SQLs executed in real databases. Valid outputs are retained through execution-aware rejection sampling, with best-of-N selection enhancing reliability. This agentic cycle uses execution feedback to govern data collection, producing higher-quality training signals without manual effort.

(c) Preference Collection via Execution Feedback Rejection Sampling: To further incorporate execution feedback, we distinguish failure types and extract preference pairs—valid versus invalid SQLs—based on their execution results. These are used to train the model with DPO, which guides learning toward executable outputs. This procedure aligns with offline reinforcement learning, leveraging historical execution trajectories to improve model behavior.

We summarize our contributions as follows:

- We propose an agentic data generation loop that combines LLM-based SQL generation, execution-aware rejection sampling, and iterative self-refinement to construct high-quality dialect-specific training data with minimal manual labeling.
- We introduce an offline reinforcement learning framework that captures execution-based preference signals and applies DPO to align the model toward generating executable SQL.
- We conduct extensive evaluations across diverse SQL dialects (PostgreSQL, MySQL, and Oracle) $\times$ difficulty levels (single domain, cross-domain, extensive database), demonstrating significant improvements over strong

baselines (e.g., GPT-4o) and providing insights for execution-guided SQL modeling.

## 2 Related Work

### 2.1 Text-to-SQL

Relational databases store a significant portion of the world's data, and retrieving information from them typically requires writing SQL queries. Automating SQL generation can lower the barrier for users to access data. A common scenario for automatic SQL generation is querying databases using natural language input (Zhong et al., 2017; Yu et al., 2018). Early research treated text-to-SQL as a semantic parsing problem, where models such as RNNs and transformer-based encoders (e.g., BERT) were trained to map natural language questions to SQL statements (Gan et al., 2021; Zhong et al., 2017; Deng et al., 2022a). Performance has also improved by incorporating additional constraints into inputs and outputs (Liu et al., 2022; Wang et al., 2021; Deng et al., 2021). With the emergence of large language models (LLMs) (Brown et al., 2020; Ouyang et al., 2022; OpenAI, 2023), text-to-SQL has been further developed using prompt-based methods and fine-tuning, benefiting from LLMs' strong instruction-following and intent understanding capabilities (Dong et al., 2023b; Li et al., 2024a; Pourreza and Rafiei, 2024; Wang et al., 2023a; Talaei et al., 2024). In practical applications, text-to-SQL has been used to handle more complex data and agent-based workflows (Lei et al., 2024; Li et al., 2024b). One challenge in real-world scenarios is handling SQL dialect differences. Early studies in domain-specific languages explored this problem using intermediate meaning representations (Guo et al., 2020). Some studies have attempted to address this issue through rule-based translation and compiler-based methods (Pourreza et al., 2024; Lin et al., 2024b).

Given the LLM-driven paradigm, this work focuses on a data-centric approach to text-to-SQL. Specifically, execution-based methods are explored to handle SQL dialect variations.

### 2.2 Code LLMs

Code foundation models have demonstrated strong code generation capabilities across various tasks. OpenAI's Codex (Chen et al., 2021a) was one of the earliest domain-specific LLMs for coding, supporting the Copilot service (Microsoft, 2023). The open-source community has further contributed with models like Deepseek-Coder (Guo et al., 2024) and StarCoder (Li et al., 2023b), which were trained from scratch on massive code-related datasets. While others, like Code-Llama (Roziere et al., 2023) and Code-Qwen (Hui et al., 2024), adapted general-purpose models through post-training on code-specific corpora. Beyond foundation models, researchers have fine-tuned them for specific applications. Maigcoder (Wei et al., 2023) enhances instruction-following abilities using curated code snippets, while Wizard-Coder (Luo et al., 2024) and WavCoder (Yu et al., 2023) refine instruction-code alignment via evol-instruct (Xu et al., 2023). OctoCoder (Muennighoff et al., 2023) leverages Git commits to enhance model adaptability. Additionally, approaches like IRCoder (Paul et al., 2024) and UniCoder (Sun et al., 2024) explore intermediate representations (e.g., LLVM) to improve code generation.

Compared to these approaches, our work also focuses on code generation but emphasizes leveraging execution signals from database environment. From the perspective of code LLM development, this approach provides insights applicable to broader code generation tasks. The Dialect SQL scenario serves as a practical testbed, allowing for clearer validation of method effectiveness.

### 2.3 Data Synthesis

Modern machine learning methods typically require large-scale and high-quality datasets (Zhou et al., 2023b) for effective learning. However, obtaining high-quality data for every corner case is often impractical, leading researchers to explore dataset generation. By integrating existing incomplete data with the extensive knowledge embedded in LLMs, data generation can produce more comprehensive datasets for model training (Wang et al., 2023b; Xu et al., 2023; Wei et al., 2023). Recently, to enhance the reasoning capabilities of LLMs, particularly in math and code, many approaches have incorporated verifiers, such as answer or reward models, to curate high-quality datasets for model refinement (Yuan et al., 2023; Guo et al., 2025; Zelikman et al., 2022).

Our work focuses on SQL execution verification. By utilizing execution results, we obtain high-quality data by rejection sampling and further refine the model through self-taught training.

## 3 Methodology

In this section, we present the details of our approach to obtain **ExeSQL**, including 3 phases: Translation Bootstrapping, Iterative Data Generation and Training, and Preference Enhancement. The key idea of Execution-Assisted Generation is fully leveraging execution verification signals to asisst LLM to generate high-quality data for text-to-SQL across different dialects. An illustration of **ExeSQL** is shown in Figure 3.

### 3.1 Formulation

We denote a natural language query as $Q$, its corresponding SQL as $S$, and the generation model as an LLM $M_\theta$. The training set $\mathcal{D} = \{(Q_i, S_i)\}_{i=1}^{N}$ is constructed by translating a high-resource source dialect $\mathcal{D}_{\text{Source}}$ (e.g., SQLite) to target dialects using a bootstrapping model and a dialect mapping function $T$.

To guide model training, we define an execution-based reward function $\mathcal{R}(S) \in \{0, 1\}$, which returns 1 if the SQL executes successfully. The goal is to train a model that maximizes expected execution success:

$$\pi_\theta^* = \arg\max_{\pi_\theta} \mathbb{E}_{Q\sim\mathcal{D}} \left[ \mathbb{E}_{\hat{S}\sim\pi_\theta(\cdot|Q)} \left[ \mathcal{R}(\hat{S}) \right] \right] \quad (1)$$

We adopt a self-evolving offline training strategy (Zelikman et al., 2022; Dong et al., 2023a; Gülçehre et al., 2023; Schulman et al., 2017), which iteratively (1) filters generated SQLs via **execution-guided rejection sampling**, and (2) applies **preference optimization** through Direct Preference Optimization (DPO). The model is updated at iteration $t$ as:

$$\pi_\theta^{(t+1)} = \arg\max_{\pi_\theta} \mathbb{E}_{Q,\hat{S},S^*\sim\mathcal{D}} \left[ \mathcal{R}(S^*, \hat{S}) \right] \quad (2)$$

Here, $S^*$ denotes a preferred (e.g., executable) SQL, contrasted against a failed candidate $\hat{S}$. This defines an offline reinforcement learning loop grounded in execution feedback.

### 3.2 Translation-based Bootstrapping

Let $D_{\text{SQLite}} = \{(Q_i, S_i)\}_{i=1}^{N}$ be a large-scale dataset containing natural language questions $Q_i$ paired with corresponding SQL queries $S_i$ written in SQLite dialect. Given the scarcity of multi-dialect SQL datasets, we first leverage $D_{\text{SQLite}}$ to bootstrap an initial dataset for training.

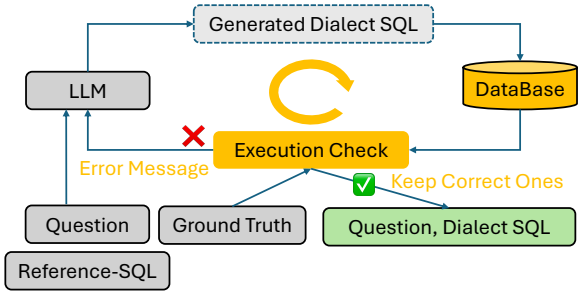


Figure 2: Execution-based error feedback loop for dialect-specific SQL refinement. Through this, we can collect a bootstrap dataset to resolve the cold-start issue of training expert dialect model.

To achieve this, we introduce a translation function $T : S_{\text{SQLite}} \rightarrow S_{\text{Target}}$, which generates an SQL query $S_{\text{Target}}$ in the target dialect based on both the original SQL query $S_{\text{SQLite}}$ and the corresponding question $Q$, modeled as:

$$S_{\text{Target}} \sim P(S_{\text{Target}}|Q, S_{\text{SQLite}})$$

However, direct translation does not guarantee correctness due to differences in SQL syntax and execution semantics across dialects. To refine the generated SQL queries, we incorporate an **execution-based verification and iterative correction mechanism**, as illustrated in Figure 2.

The refinement process operates as follows (Appendix A.13): 1) An LLM (GPT-4o here) generates candidate SQL queries $S_{\text{Target}}$ for a given natural language question $Q$, conditioned on $S_{\text{SQLite}}$. 2) The generated SQL query is executed in a database corresponding to the target dialect. 3) If the execution succeeds, the query is added to the validated dataset: $D_{\text{Trans}} = \{(Q_i, S_{\text{Target},i})\}$ 4) If the execution fails, the database returns an error message, which is fed back into the LLM as an additional context for refining the SQL query. The model iteratively refines $S_{\text{Target}}$ until a valid query is produced. 5) This iterative execution check continues until either a valid SQL query is found or a maximum refinement threshold is reached.

This approach effectively corrects syntactic and semantic errors by leveraging real execution feedback rather than relying solely on static rule-based translation. Through this execution-aware iteration, the model progressively learns to generate more accurate and dialect-specific SQL queries. The final dataset, $D_{\text{Trans}}$, serves as a high-quality dialect training corpus, enabling robust generalization across different database systems.

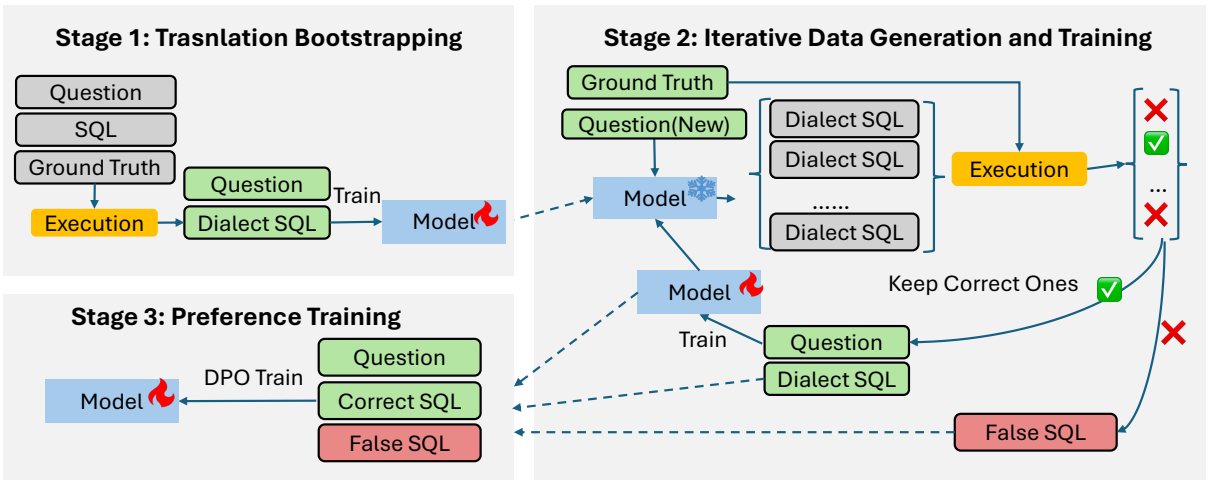


Figure 3: Pipeline for Dialect Text-to-SQL Data Generation and Model Training. The framework consists of three stages: (1) **Translation Bootstrapping**: A bootstrap text-to-SQL model is fine-tuned using SQL translations from an existing dataset (e.g., SQLite) to other dialects (e.g., MySQL, PostgreSQL). (2) **Iterative Data Generation and Training**: The model generates multiple SQL candidates per question, which are validated via execution feedback. Correct queries are retained to refine the dataset, enabling iterative self-improvement. (3) **Preference Enhancement**: A Direct Preference Optimization (DPO) step is applied to distinguish correct and incorrect SQL queries. High-quality pairs (question, correct SQL) are used to further improve the model's performance and preference learning, ensuring both correctness and efficiency in SQL generation.

### 3.3 Iterative Data Generation and Training

While $D_{\text{Trans}}$ provides a baseline, rule-based translation alone is insufficient to guarantee correctness due to syntax differences, type constraints, and execution behaviors across SQL dialects. To address this, we introduce an iterative execution-feedback process incorporating rejection sampling and augmented question generation, as depicted in Figure 3.

#### 3.3.1 Augmenting Training Data with New Questions

To improve model generalization across SQL dialects, we incorporate additional natural language questions from two sources:**(1) Existing Text-to-SQL Datasets**: We extract additional questions from existing datasets like WikiSQL, ensuring coverage of diverse query structures. **(2) Database-Aware Question Generation**: We leverage **GPT-4o** to generate new questions based on actual database values. Given a schema and sample database records, GPT-4o generates contextually relevant questions that reference specific values, improving the model's robustness in handling real-world queries.

By integrating these new questions, we expand our dataset beyond simple rule-based translations, allowing the model to generate and validate SQL queries for a more diverse set of inputs.

#### 3.3.2 Execution-based Rejection Sampling

For each natural language question $Q_i$, the model $M_\theta$ generates multiple dialect-specific SQL candidates $\{S_{\text{cand},i}\}$, following the probability distribution:$S_{\text{cand},i} \sim P_\theta(S|Q_i)$

Each candidate query is then executed in the corresponding database environment, yielding an execution result $R(S_{\text{cand},i})$:

$$R(S) = \begin{cases} 1, & \text{if } S \text{ executes successfully} \\ 0, & \text{if } S \text{ fails due to execution errors} \end{cases}$$

We apply a rejection sampling to iteratively refine SQL generation: If $S_{\text{cand}}$ exectues successfully, i.e., $R(S_{\text{cand},i}) = 1$. The query is added to the validated dataset: $D_{\text{Valid}} = D_{\text{Valid}} \cup \{(Q_i, S_{\text{cand},i})\}$

If $S_{\text{cand}}$ is a Failure Case, i.e., $R(S_{\text{cand},i}) = 0$. The query is stored in the negative dataset: $D_{\text{Neg}} = D_{\text{Neg}} \cup \{(Q_i, S_{\text{cand},i})\}$

This process is iteratively repeated until a valid SQL query is generated or a predefined iteration limit is reached.

#### 3.3.3 Iterative Data Generation and Model Refinement

The validated dataset $D_{\text{Valid}}$ is used for further fine-tuning, while incorrect queries in $D_{\text{Neg}}$ serve as contrastive learning signals in later preference optimization stages.

This process results in a high-quality, dialect-aware text-to-SQL dataset that is continuously refined through execution-based validation and real-world query augmentation.

### 3.4 Preference Optimization

To further refine the model's SQL generation capabilities, we leverage DPO (Rafailov et al., 2023) to distinguish between correct and incorrect queries, using execution feedback as the primary signal. The negative dataset $D_{\text{Neg}}$ and validated dataset $D_{\text{Valid}}$ have already been collected during the Iterative Data Generation and Training phase. Here, we construct preference pairs to fine-tune the model based on execution outcomes.

**Pairwise Preference Data Construction** To enable preference learning, we form query pairs $(S_{\text{pos}}, S_{\text{neg}})$, where: $S_{\text{pos}} \in D_{\text{Valid}}, \quad S_{\text{neg}} \in D_{\text{Neg}}$ These pairs allow the model to differentiate between correct and incorrect SQL, ensuring that preference learning reinforces correct generation.

**Direct Preference Optimization (DPO) Training** The model is fine-tuned using DPO, where the objective is to maximize the probability of generating preferred SQL queries over non-preferred ones:

$$P_\theta(S_{\text{pos}}|Q) > P_\theta(S_{\text{neg}}|Q)$$

By leveraging execution failures as negative examples and correct executions as positive examples, the model learns to generate more reliable and executable SQL queries. This approach enhances both the correctness and robustness of SQL generation across different dialects.

## 4 Implementation and Evaluation Settings

The bootstrap dataset and new questions for **ExeSQL** are generated using GPT-4o (OpenAI, 2023). We choose GPT-4o due to its superior ability to follow instructions and leverage error messages to generate accurate bootstrap dialect SQL examples. The final **ExeSQL** dataset consists of 20.6k samples in the supervised finetuning (SFT) set and 8k samples in the preference pairs (Appendix A.2). All training is conducted on four A6000 GPUs. We fine-tune the full-parameter Deepseek-Coder-7B (Guo et al., 2024) for supervised finetuning (SFT) and Direct Preference Optimization (DPO). For detailed training configurations and inference hyperparameters, please refer to Appendix A.3

For baseline comparisons, we evaluate GPT-4o-2024-11-20 and Gemini-1.5-pro-0827 (Reid et al., 2024), both of which were released in 2024. Since these models were trained on publicly available data up to their release dates, they likely include extensive SQL-related training data, ensuring a fair comparison.

### 4.1 Text-to-SQL across dialects and Benchmarks

**Dialects.** To fully validate the generalization ability of our method, we selected three SQL dialects: `PostgreSQL`, `MySQL` and `Oracle`. Our pipeline is dialect-agnostic, we chose these two dialects to verify the generalizable effectiveness of our pipeline across different dialects.

**Benchmarks.** We adapt three standard benchmarks, Spider (Yu et al., 2018) WikiSQL (Zhong et al., 2017) and BIRD (Li et al., 2024b), for in-domain evaluation and use Dr.Spider (Chang et al., 2023) as an out-of-domain dataset. We also incorporate the single-domain benchmark MimicSQL (Wang et al., 2020; Deng et al., 2022b) to evaluate our model across varying difficulty levels. For dialect SQL evaluation, we extract the question, database, and ground truth result, prompting the model to generate dialect-specific SQL and verifying execution accuracy. Details on these datasets are in Appendix A.9. To ensure accurate evaluation, we preprocess responses to extract SQL using an answer extraction tool (Appendix A.12). For results on the single-domain dataset, please refer to Appendix A.8.

### 4.2 Baseline Models

**General purposed LLM baselines:** We evaluate four large language models (LLMs) without any fine-tuning for text-to-SQL tasks: `GPT-4o` (OpenAI, 2023), `Gemini-1.5-pro` (Reid et al., 2024), and `Llama3.1-Instruct` (met). These models are assessed by directly prompting them to generate SQL queries given a natural language question and the corresponding database schema.

**Code Expert LLM baselines:** These baselines consist of LLMs trained on large-scale code-related corpora, making them well-suited for code generation tasks. We include `DeepSeek-Coder` (Guo et al., 2024), `Qwen-Coder` (Hui et al., 2024), `Magicoder-DS` (Wei et al., 2023), and `WizardCoder` (Luo et al., 2024).

**SQL Expert LLM baselines:** Several LLMs are

| Method | Model size | PostgreSQL | | MySQL | | | Oracle | Average |
|---|---|---|---|---|---|---|---|---|
| | | Spider | WikiSQL | Spider | WikiSQL | Bird | Spider | |
| *General purposed LLM* | | | | | | | | |
| **GPT-4o** | - | 54.59 | 58.97 | 62.09 | 57.24 | 36.38 | 64.86 | 55.69 |
| **Gemini-1.5-pro** | - | 51.03 | 54.1 | 64.90 | 51.95 | 36.11 | 65.21 | 53.88 |
| **Llama3.1-Instruct** | 8B | 33.63 | 31.6 | 48.86 | 25.41 | 24.58 | 30.0 | 32.35 |
| *Code Expert LLM* | | | | | | | | |
| **Deepseek-Coder** | 7B | 37.31 | 18.12 | 49.6 | 24.67 | 16.00 | 50.77 | 32.75 |
| **Qwen-Coder** | 7B | 36.8 | 15.48 | 39.04 | 22.84 | 15.36 | 58.31 | 31.31 |
| **Magicoder** | 7B | 21.9 | 17.45 | 47.28 | 23.32 | 13.23 | 26.6 | 24.96 |
| **WizardCoder** | 15B | 23.78 | 16.91 | 32.36 | 20.56 | 18.38 | 36.33 | 24.72 |
| *SQL Expert LLM* | | | | | | | | |
| **CodeS** | 7B | 24.76 | 20.0 | 35.6 | 23.0 | 14.41 | 37.4 | 25.86 |
| **StructLLM** | 7B | 38.71 | 30.97 | 44.2 | 7.14 | 22.69 | 33.16 | 29.48 |
| **ExeSQL** | 7B | **69.86** | **74.10** | **72.09** | **73.64** | **41.13** | **69.35** | **66.70** |

Table 1: Performance comparison of various LLMs on Dialect text-to-SQL benchmarks. **ExeSQL** surpasses all baseline models, achieving an average improvement of 11.0% over GPT-4o.

specifically adapted for SQL generation, typically optimized for the SQLite dialect and demonstrating strong table understanding capabilities. We include `Code-S` (Li et al., 2024a) and `StructLLM` (Zhuang et al., 2024) in this category.

The comparisons in (2) and (3) aim to assess whether fine-tuned general-purpose LLMs can outperform specialized code-generation or SQL-focused models in specific scenarios.

## 5 Experimental Results

### 5.1 Main Results

We present the main experimental results in Table 1. From the table, we observe that **ExeSQL** achieves an average accuracy of 66.70% across PostgreSQL, MySQL and Oracle benchmarks, significantly outperforming all baseline models.

**General purposed LLMs.** Among the general-purpose LLMs, GPT-4o achieves the highest accuracy (55.69%), demonstrating its strong zero-shot SQL generation capability. We find that Gemini-1.5-pro underperforms GPT-4o, achieving 53.88%. Llama3.1-8B-Instruct perform worse, with average accuracies of 32.35%, respectively. These results indicate that general-purpose LLMs struggle with SQL dialect variations.

**Code Expert LLMs.** Code-focused models, such as Deepseek-Coder and Qwen-Coder, demonstrate better performance than standard LLMs. Deepseek-Coder achieves an average accuracy of 32.75%, while Qwen-Coder reaches 31.31%. However, Magicoder and WizardCoder perform worse, suggesting that general code generation ability does not equal SQL generation (especially dialect) capability. This implies that code training alone is insufficient for SQL dialect adaptation.

**SQL Expert LLMs.** The SQL-specialized models exhibit the most significant improvements. StructLLM, which is trained on SQL-specific tasks, achieves an accuracy of 29.48%, slightly outperforming most code models. However, **ExeSQL** surpasses all baselines by a large margin, reaching an average accuracy of 66.70%. Also, it is worth noting that these models often have a great performance degradation compared with SQLite performance (Appendix A.1).

These results highlight the importance of the proposed execution-based fine-tuning and dialect-aware SQL adaptation. Unlike general-purpose or code-focused models, **ExeSQL** effectively learns to handle different SQL dialects through iterative refinement, leading to a substantial performance boost.

### 5.2 Further Analysis

To validate the effectiveness of **ExeSQL**, we conduct three analyses: (1) Ablation studies assess the impact of iterative refinement and preference learning on accuracy. (2) ID and OOD evaluation measures generalization to unseen queries and SQL dialects. (3) Execution-based rejection sampling analysis examines its role in improving SQL correctness. These analyses confirm **ExeSQL**'s

robustness and adaptability.

| Method | PostgreSQL | MySQL |
|---|---|---|
| ExeSQL | 71.98 | 72.865 |
| w/o iteration | 63.49 | 60.09 |
| w/o preference | 71.36 | 70.34 |

Table 2: Performance comparison of different ExeSQL ablations.

#### 5.2.1 Ablations for Iteration data generation.

Table 2 shows that removing iteration-based refinement significantly reduces performance (**71.98%** to **63.49%** on PostgreSQL, **72.865%** to **60.09%** on MySQL), highlighting the importance of iterative data generation in improving SQL accuracy. Removing preference learning also leads to a performance drop, though less severe, indicating that preference optimization further refines query quality. These results demonstrate that both iterative refinement and preference learning play crucial roles in enhancing **ExeSQL**'s effectiveness.

| Method | PostgreSQL | | MySQL | |
|---|---|---|---|---|
| | Spider | Dr. | Spider | Dr. |
| Deepseek-Coder | 37.31 | 27.10 | 49.6 | 36.82 |
| StructLLM | 38.71 | 25.83 | 44.2 | 40.00 |
| ExeSQL | **69.86** | **59.16** | **72.09** | **56.02** |

Table 3: Results on ID and OOD evaluation. The strong and consistent performance of **ExeSQL** demonstrates its robust generalization ability without signs of overfitting.

#### 5.2.2 ID and OOD Evaluation.

We evaluate **ExeSQL** on both in-distribution (ID) and out-of-distribution (OOD) datasets to assess its generalization. The OOD evaluation is conducted on **Dr.Spider** (Chang et al., 2023), a diagnostic text-to-SQL benchmark with 15,269 samples, introducing perturbations in databases (DB), natural language queries (NLQ), and SQL to test robustness. Given its scale, Dr.Spider is significantly harder to overfit than Spider's 2,147 samples.

Table 3 shows that **ExeSQL** consistently achieves the highest accuracy across all settings. Notably, ExeSQL outperforms StructLLM and Deepseek-Coder by a large margin on both PostgreSQL and MySQL, confirming its strong generalization to both ID and OOD queries.

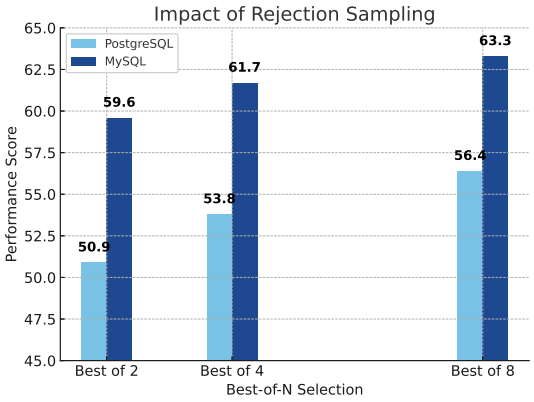


Figure 4: Retention rate of correct dialect SQL samples across different best-of-N sampling strategies, evaluated on 1,000 query samples. The results indicate that the bootstrapped model can already generate many correct dialect samples, and increasing N further improves correctness.

#### 5.2.3 Configuration of Execution-based Rejection Sampling.

Figure 4 presents the effect of execution-based rejection sampling on SQL generation accuracy across different best-of-N selection strategies. As $N$ increases, the proportion of correct dialect SQL samples improves consistently for both PostgreSQL and MySQL.

This result indicates that the bootstrapped model is capable of generating a significant number of correct dialect SQL queries even without additional fine-tuning. The primary challenge then shifts to efficiently identifying and selecting these correct samples. An iterative sampling approach can be employed to extract high-quality SQL queries, which can further enhance the model through self-supervised training.

## 6 Conclusion

We propose an execution-driven framework to enhance text-to-SQL generation across multiple SQL dialects. By integrating LLM-based dialect bootstrapping, execution feedback rejection sampling, and preference learning, our approach iteratively refines SQL generation through execution validation and error correction. Experiments show that **ExeSQL** outperforms GPT-4o by a large margin on 2 dialects, respectively, demonstrating superior adaptability and correctness. Our findings highlight the importance of execution-aware training and provide a scalable solution for robust multi-dialect text-to-SQL modeling.

## Limitations

In this work, we primarily focus on two mainstream dialects (MySQL and PostgreSQL) within a relatively simple environment. This setting overlooks complexities that arise in larger-scale or heterogeneous scenarios, and it only partially addresses advanced dialect-specific features (e.g., complex window functions or Regex handling). Moreover, our iterative generation process relies on predefined prompts and partial rules, which may not readily accommodate databases with significantly different formal grammars. In future research, we plan to explore more dialects and more complex database conditions, aiming to enhance the coverage and robustness of our multi-dialect text-to-SQL framework.

## References


Meta llama 3. https://ai.meta.com/blog/meta-llama-3/. Accessed: 2024-06-10.

Anton Alexandrov, Veselin Raychev, Mark Niklas Müller, Ce Zhang, Martin Vechev, and Kristina Toutanova. 2024. Mitigating catastrophic forgetting in language transfer via model merging. *Preprint*, arXiv:2407.08699.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, and 12 others. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Shuaichen Chang, Jun Wang, Mingwen Dong, Lin Pan, Henghui Zhu, Alexander Hanbo Li, Wuwei Lan, Sheng Zhang, Jiarong Jiang, Joseph Lilien, and 1 others. 2023. Dr. spider: A diagnostic evaluation benchmark towards text-to-sql robustness. *arXiv preprint arXiv:2301.08881*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021a. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021b. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Naihao Deng, Yulong Chen, and Yue Zhang. 2022a. Recent advances in text-to-sql: A survey of what we have and what we expect. In *Proceedings of the 29th International Conference on Computational Linguistics, COLING 2022, Gyeongju, Republic of Korea, October 12-17, 2022*, pages 2166–2187. International Committee on Computational Linguistics.

Naihao Deng, Yulong Chen, and Yue Zhang. 2022b. Recent advances in text-to-SQL: A survey of what we have and what we expect. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 2166–2187, Gyeongju, Republic of Korea. International Committee on Computational Linguistics.

Xiang Deng, Ahmed Hassan, Christopher Meek, Oleksandr Polozov, Huan Sun, and Matthew Richardson. 2021. Structure-grounded pretraining for text-to-sql. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1337–1350.

Hanze Dong, Wei Xiong, Deepanshu Goyal, Rui Pan, Shizhe Diao, Jipeng Zhang, Kashun Shum, and Tong Zhang. 2023a. RAFT: reward ranked finetuning for generative foundation model alignment. *CoRR*, abs/2304.06767.

Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Jinshu Lin, Dongfang Lou, and 1 others. 2023b. C3: Zero-shot text-to-sql with chatgpt. *arXiv preprint arXiv:2307.07306*.

Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R. Woodward, John Drake, and Qiaofu Zhang. 2021. Natural SQL: Making SQL easier to infer from natural language specifications. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2030–2042, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Çaglar Gülçehre, Tom Le Paine, Srivatsan Srinivasan, Ksenia Konyushkova, Lotte Weerts, Abhishek Sharma, Aditya Siddhant, Alex Ahern, Miaosen Wang, Chenjie Gu, Wolfgang Macherey, Arnaud Doucet, Orhan Firat, and Nando de Freitas. 2023. Reinforced self-training (rest) for language modeling. *CoRR*, abs/2308.08998.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets

programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Jiaqi Guo, Qian Liu, Jian-Guang Lou, Zhenwen Li, Xueqing Liu, Tao Xie, and Ting Liu. 2020. Benchmarking meaning representations in neural semantic parsing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 1520–1540. Association for Computational Linguistics.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida I. Wang, and Tao Yu. 2024. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *CoRR*, abs/2411.07763.

Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024a. Codes: Towards building open-source language models for text-to-sql. *Proceedings of the ACM on Management of Data*, 2(3):1–28.

Jinyang Li, Binyuan Hui, GE QU, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023a. Can LLM already serve as a database interface? a BIg bench for large-scale database grounded text-to-SQLs. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, and 1 others. 2024b. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023b. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Yong Lin, Hangyu Lin, Wei Xiong, Shizhe Diao, Jianmeng Liu, Jipeng Zhang, Rui Pan, Haoxiang Wang, Wenbin Hu, Hanning Zhang, Hanze Dong, Renjie Pi, Han Zhao, Nan Jiang, Heng Ji, Yuan Yao, and Tong Zhang. 2024a. Mitigating the alignment tax of rlhf. *Preprint*, arXiv:2309.06256.

Zhisheng Lin, Yifu Liu, Zhiling Luo, Jinyang Gao, and Yu Li. 2024b. Momq: Mixture-of-experts enhances multi-dialect query generation across relational and non-relational databases. *arXiv preprint arXiv:2410.18406*.

Qian Liu, Bei Chen, Jiaqi Guo, Morteza Ziyadi, Zeqi Lin, Weizhu Chen, and Jian-Guang Lou. 2022. TAPEX: table pre-training via learning a neural SQL executor. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. Wizardcoder: Empowering code large language models with evol-instruct. *International Conference on Learning Representations (ICLR)*.

Toby Mao. 2023. Sqlglot. `https://github.com/tobymao/sqlglot`. Accessed: 2024-06-09.

Microsoft. 2023. Github copilot – your ai pair programmer. GitHub repository.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*.

OpenAI. 2023. GPT-4 technical report. *CoRR*, abs/2303.08774.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Rui Pan, Jipeng Zhang, Xingyuan Pan, Renjie Pi, Xiaoyu Wang, and Tong Zhang. 2024. Scalebio: Scalable bilevel optimization for llm data reweighting. *Preprint*, arXiv:2406.19976.

Indraneil Paul, Jun Luo, Goran Glavaš, and Iryna Gurevych. 2024. Ircoder: Intermediate representations make language models robust multilingual code generators. *arXiv preprint arXiv:2403.03894*.

Mohammadreza Pourreza and Davood Rafiei. 2024. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Advances in Neural Information Processing Systems*, 36.

Mohammadreza Pourreza, Ruoxi Sun, Hailong Li, Lesly Miculicich, Tomas Pfister, and Sercan Ö. Arik. 2024. SQL-GEN: bridging the dialect gap for text-to-sql via synthetic data and model merging. *CoRR*, abs/2408.12733.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36:53728–53741.

Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy Lillicrap, Jean-baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, and 1 others. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Services. 2023. A. w. ai code generator - amazon codewhisperer - aws. Amazon Page.

Tao Sun, Linzheng Chai, Jian Yang, Yuwei Yin, Hongcheng Guo, Jiaheng Liu, Bing Wang, Liqun Yang, and Zhoujun Li. 2024. Unicoder: Scaling code large language model via universal code. *arXiv preprint arXiv:2406.16441*.

Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. Chess: Contextual harnessing for efficient sql synthesis. *arXiv preprint arXiv:2405.16755*.

Bailin Wang, Mirella Lapata, and Ivan Titov. 2021. Learning from executions for semantic parsing. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2747–2759, Online. Association for Computational Linguistics.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Qian-Wen Zhang, Zhao Yan, and Zhoujun Li. 2023a. Mac-sql: Multi-agent collaboration for text-to-sql. *arXiv preprint arXiv:2312.11242*.

Ping Wang, Tian Shi, and Chandan K. Reddy. 2020. Text-to-sql generation for question answering on electronic medical records. *Preprint*, arXiv:1908.01839.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language model with self generated instructions. *arXiv preprint arXiv:2212.10560*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023b. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 13484–13508. Association for Computational Linguistics.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023c. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL) (Volume 1: Long Papers)*, pages 13484–13508.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, and 3 others. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Sang Michael Xie, Hieu Pham, Xuanyi Dong, Nan Du, Hanxiao Liu, Yifeng Lu, Percy Liang, Quoc V. Le, Tengyu Ma, and Adams Wei Yu. 2023. Doremi: Optimizing data mixtures speeds up language model pretraining. *Preprint*, arXiv:2305.10429.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, and 1 others. 2018. Spider: A large-scale human-labeled dataset for complex and

cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2023. Wavecoder: Widespread and versatile enhanced instruction tuning with refined data generation. *arXiv preprint arXiv:2312.14187*.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. 2023. Scaling relationship on learning mathematical reasoning with large language models. *arXiv preprint arXiv:2308.01825*.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. 2022. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand. Association for Computational Linguistics.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103.

Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, Susan Zhang, Gargi Ghosh, Mike Lewis, Luke Zettlemoyer, and Omer Levy. 2023a. Lima: Less is more for alignment. *Preprint*, arXiv:2305.11206.

Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, and 1 others. 2023b. Lima: Less is more for alignment. *arXiv preprint arXiv:2305.11206*.

Alex Zhuang, Ge Zhang, Tianyu Zheng, Xinrun Du, Junjie Wang, Weiming Ren, Stephen W Huang, Jie Fu, Xiang Yue, and Wenhu Chen. 2024. Structlm: Towards building generalist models for structured knowledge grounding. *arXiv preprint arXiv:2402.16671*.

Ran Zmigrod, Salwa Alamir, and Xiaomo Liu. 2024. Translating between sql dialects for cloud migration. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, pages 189–191.

# A Appendix

## A.1 Analysis for the Dialect-Degradation for Text-to-SQL

Table 4 demonstrates that general LLMs experience significant performance degradation across SQL dialects, with Flip Δ ranging from -12.83 to -47.72. Larger models such as GPT-4o and Gemini-1.5-pro degrade less, while smaller models like CodeS and StructLLM suffer more. In contrast, SQL-Expert models exhibit even 2-3× higher degradation, likely due to weaker generalization from smaller parameter sizes. This highlights the importance of SQL dialect adaptation research, as even strong general LLMs struggle with dialect shifts.

| **Method** | **PostgreSQL** | **MySQL** | **SQLite** | Δ |
|---|---|---|---|---|
| GPT-4o | 54.59 | 62.09 | 71.17 | -12.83 |
| Gemini-1.5-pro | 51.03 | 64.90 | 77.27 | -19.31 |
| CodeS | 24.76 | 35.60 | 77.90 | -47.72 |
| StructLLM | 38.71 | 44.20 | 70.20 | -28.75 |

Table 4: Zero-shot performance comparison on Spider across different SQL dialects. Flip Δ measures the differences between the model's performance on SQlite compared with PostgreSQL and MySQL.

## A.2 Generated Data Statistics

Table 5 shows the distribution of question sources. During SFT data generation, 3.7k new dialect-specific questions were created based given the values in databases.

| Stage | Dataset | Size |
|---|---|---|
| SFT | Spider | 6.9k |
| | WikiSQL | 10k |
| | New Generated Data | 3.7k |
| DPO | Spider | 4k |
| | WikiSQL | 4k |

Table 5: Dataset statistics for SFT and DPO training stages.

## A.3 Implementation

We fine-tune the full-parameter Deepseek-Coder-7B (Guo et al., 2024) using supervised finetuning (SFT) for one epoch with a batch size of 16 and a learning rate of 2e-5. For Direct Preference Optimization (DPO) training, we train for three epochs with a batch size of 16 and a learning rate of 5.0e-6. Additionally, we incorporate the SFT loss into the DPO loss with a weight of 1 during preference training. For execution-based rejection sampling and worst-of-n negative sample collection, we set the inference parameters to temperature = 0.7, top-p = 0.9, and top-k = 50. Negative examples for

DPO training are selected using the worst-of-N strategy, with N = 8.

Here, we make use of huggingface-transformer (Wolf et al., 2020) and llama-factory (Zheng et al., 2024) to perform training. For the inference, we make use of vllm toolkit (Kwon et al., 2023).

| Setting | Value |
|---|---|
| GPUs Used | 4 × A6000 |
| Model | Deepseek-Coder-7B |
| Batch Size | 16 |
| Epochs | 1 |
| Learning Rate | $2 \times 10^{-5}$ |

Table 6: Supervised Fine-tuning (SFT) Configuration

| Setting | Value |
|---|---|
| GPUs Used | 4 × A6000 |
| Model | Deepseek-Coder-7B |
| Batch Size | 16 |
| Epochs | 3 |
| Learning Rate | $5 \times 10^{-6}$ |
| Loss Weight | 1 (SFT + DPO) |
| Worst-of-N | 8 |

Table 7: Direct Preference Optimization (DPO) Configuration

| Setting | Value |
|---|---|
| Temperature | 0.7 |
| Top-p | 0.9 |
| Top-k | 50 |

Table 8: Inference and Rejection Sampling Configuration

### A.4 Impact of Direct Preference Optimization (DPO)

To understand the performance impact of Direct Preference Optimization (DPO) following Supervised Fine-tuning (SFT), we evaluated the performance on both PostgreSQL and MySQL dialects. The results, based on an initial SFT with 8,000 samples and a subsequent DPO with 20,000 preference samples, are presented in Table 9.

| Model | PostgreSQL | MySQL | Oracle |
|---|---|---|---|
| SFT | 71.36 | 70.34 | 65.86 |
| SFT + DPO | 71.98 | 72.86 | 69.35 |

Table 9: Performance Comparison: SFT vs. SFT + DPO

Furthermore, to analyze the detailed effect of DPO on model robustness and generalization, we evaluated both models' performance under database perturbation and SQL perturbation. The results, presented as the average performance across PostgreSQL and MySQL, are shown in Table 10.

| Model | Database Perturbation | SQL Perturbation | Average |
|---|---|---|---|
| SFT | 55.54 | 62.16 | 58.85 |
| SFT + DPO | 57.14 | 64.06 | 60.60 |

Table 10: Robustness and Generalization Analysis: SFT vs. SFT + DPO (Average)

These results suggest that DPO enhances **generalization**, especially on unseen or more diverse test cases, even with a relatively small amount of preference training data. From a data perspective, this aligns with the core insight of LIMA (Zhou et al., 2023a)—that a few high-quality preference samples can be highly effective for alignment.

In our case, although the total amount of DPO data is limited (8k preference pairs derived from 20k samples), it still results in noticeable improvements in both robustness and generalization for text-to-SQL tasks. This underscores that data quality and diversity are key to effective model tuning. We attribute the quality of our DPO data to the execution-based verification method used during preference construction.

### A.5 Impact of Data Translation and Augmentation Strategies

To clarify the impact of different data translation and augmentation strategies on the performance of our Supervised Fine-tuning (SFT) baselines, we provide a comparison of SFT results under three distinct approaches:

- **Translation (once):** One-pass LLM translation without any further refinement (as described in Section 3.2 of the main paper).
- **Translation (iterative):** LLM translation enhanced with execution feedback within an it-

erative loop (as detailed in Section 3.2 of the main paper).

- **Translation + Augmented data:** Combines the translated data with newly generated question-SQL pairs derived from table rows (as described in Section 3.3 of the main paper). The **Final** setting integrates iterative refinement, data augmentation, and Direct Preference Optimization (DPO).

The performance of these strategies on the PostgreSQL and MySQL dialects is summarized in Table 11.

| Strategy | PostgreSQL | MySQL |
|---|---|---|
| Translation (once) | 63.49 | 60.09 |
| Translation (iterative w/ execution) | 69.97 | 67.63 |
| Translation + Augmented data | 71.36 | 70.34 |
| Final | 71.98 | 72.86 |

Table 11: Impact of Data Translation and Augmentation Strategies on Performance

These results clearly demonstrate the significant benefits of incorporating execution feedback in the translation process and further enhancing the training data through augmentation techniques. The "Final" setting, which combines iterative refinement, data augmentation, and DPO, achieves the highest performance on both PostgreSQL and MySQL.

### A.6 Impact of Multi-Dialect Training

To investigate how well Large Language Models (LLMs) can learn from training data across different SQL dialects and the potential benefits of multi-dialect training, we conducted a Supervised Fine-tuning (SFT) experiment using the Spider dataset with two primary dialects: PostgreSQL and MySQL. We evaluated the cross-dialect generalization performance under three different training settings:

- **Only PostgreSQL:** Model trained on Spider data augmented with PostgreSQL-specific syntax.
- **Only MySQL:** Model trained on Spider data augmented with MySQL-specific syntax.
- **Mixed Training:** Model trained on a dataset combining both PostgreSQL and MySQL augmented Spider data.

The results of this experiment are summarized in Table 12.

| Training Setting | PostgreSQL | MySQL | Average |
|---|---|---|---|
| Only PostgreSQL | 74.94 | 58.30 | 66.62 |
| Only MySQL | 59.94 | 74.96 | 67.45 |
| Mixed Training | 72.15 | 68.61 | 70.38 |

Table 12: Cross-Dialect Generalization Performance

We observed two interesting trends from these results:

- Models tend to overfit to the specific syntax they are trained on, resulting in a significant performance drop when evaluated on a different dialect.
- Although a naive mixed training approach improves the overall average performance, it slightly reduces the peak performance achieved on individual dialects when trained solely on that dialect.

We hypothesize that this phenomenon is related to "forgetting" (Kirkpatrick et al., 2017; Alexandrov et al., 2024). To further improve cross-dialect generalization, more sophisticated mixing strategies such as in-batch mixing (Pan et al., 2024; Xie et al., 2023), data replay (Lin et al., 2024a), or even model merging (Alexandrov et al., 2024) may be necessary.

### A.7 Empirical Analysis of Data Diversity

We present an empirical experiment designed to investigate the data diversity of our generated samples compared to a baseline. As discussed in the main body of the paper, the validity verification mechanism based solely on SQL execution correctness during the data synthesis phase may suffer from dimensionality limitations. This can potentially lead Large Language Models (LLMs) to generate structurally homogeneous question-answer (Q&A) pairs, consequently reducing data diversity.

To address this potential diversity collapse, our generative iteration process explicitly incorporates diversity by varying the prompts with in-context exemplars at every generation round, a strategy similar to self-instruct (Wang et al., 2023c)

To empirically study the data diversity issue, we conducted the following experiment:

The results presented in Table 13 show that our generated data is more diverse than the baseline Spider samples, suggesting that our multi-round varying prompting strategy effectively mitigates diversity collapse.

| Dataset | Similarity Score |
|---|---|
| Our Data | 0.470 |
| Spider Sample Comparison | 0.672 |

Table 13: Comparison of Cosine Similarity Scores (TF-IDF Embeddings) between Our Generated Data and Baseline Spider Samples. Our data (10K samples paired with 5K Spider samples) shows a lower average similarity score compared to the similarity within the Spider dataset (10K Spider samples paired with highest similarity HumanEval samples), suggesting higher diversity.

On the other hand, we believe the most effective way to ensure data diversity is through access to diverse database schemas. Our method focuses on generating accurate and precise QA pairs given a particular database. As more varied databases become available, our generation framework is expected to produce even more diverse and useful training data.

### A.8 Generalization to Single-Domain Datasets

Our method is not limited by dataset type and can be applied to single-domain datasets as well. To demonstrate this, we applied our model to the MimicSQL dataset (Wang et al., 2020; Deng et al., 2022b) using the MySQL dialect **without further training**. The results are shown in Table 14.

| Model | Accuracy |
|---|---|
| GPT-4o | 72.87 |
| DeepSeek-Coder-7B | 63.66 |
| Qwen2.5-Coder-7B | 61.46 |
| StructLM-7B | 38.34 |
| ExeSQL (ours) | **76.07** |

Table 14: Accuracy on the MimicSQL Dataset (MySQL Dialect, Zero-Shot)

These results show that our method generalizes well to the single-domain setting, achieving a competitive accuracy of 76.07% on the MimicSQL dataset in a zero-shot manner, even outperforming larger, general-purpose models like GPT-4o. This highlights the robustness and adaptability of our approach beyond cross-domain benchmarks.

### A.9 Details of Evaluation Datasets

| Dataset | Spider | WikiSQL | Dr.Spider | Bird | MimicSQL |
|---|---|---|---|---|---|
| # samples | 2,147 | 8,421 | 15,269 | 1534 | 999 |

Table 15: Number of samples in different datasets

**Spider.** Spider provides a diverse collection of training and development samples, along with a hidden test set. The training set includes a mix of manually annotated examples and additional samples sourced from previous text-to-SQL datasets. Covering a wide range of databases across various domains, Spider serves as a comprehensive benchmark for evaluating cross-domain text-to-SQL performance. We used the test set of Spider with 2,147 examples to perform evaluation here.

**WikiSQL.** WikiSQL is a large-scale dataset consisting of natural language questions, SQL queries, and structured tables extracted from Wikipedia. It offers a well-organized set of training, development, and test examples, each containing a question, a table, an SQL query, and the expected execution result. We used the dev set of Spider with 8,421 examples to perform evaluation here.

**Dr.Spider.** Dr.Spider, an extension of Spider, introduces various perturbations across questions, databases, and SQL queries to assess the robustness of text-to-SQL models. It includes test sets designed to evaluate the impact of database modifications, question variations, and SQL transformations, making it a challenging benchmark for robustness testing. We used the perturbed set over all the questions, databases, and SQL queries with 15,269 examples to perform evaluation here. Detailed perturbation types are shown in Table 16.

**Bird.** Bird is a large-scale, challenging dataset specifically designed to evaluate the in-context learning capabilities of text-to-SQL models. It encompasses a wide variety of complex SQL queries and database schemas, demanding strong reasoning and schema understanding. The dataset includes training, development, and test splits. For our experiments, we utilized the development set of BIRD, which comprises 1,534 examples for evaluation.

**MimicSQL.** MimicSQL is a single-domain text-to-SQL dataset derived from the MIMIC-III electronic health records database. It focuses on the medical domain, presenting unique challenges related to medical terminology and complex database structures within healthcare. The dataset includes training and test sets. We performed our evaluation on the test set of MimicSQL_natural, which contains 999 examples.

| Perturb Type | # Samples |
|---|---|
| DB_DBcontent_equivalence | 382 |
| DB_schema_abbreviation | 2853 |
| DB_schema_synonym | 2619 |
| NLQ_column_attribute | 119 |
| NLQ_column_carrier | 579 |
| NLQ_column_synonym | 563 |
| NLQ_column_value | 304 |
| NLQ_keyword_carrier | 399 |
| NLQ_keyword_synonym | 953 |
| NLQ_multitype | 1351 |
| NLQ_others | 2819 |
| NLQ_value_synonym | 506 |
| SQL_comparison | 178 |
| SQL_DB_number | 410 |
| SQL_DB_text | 911 |
| SQL_NonDB_number | 131 |
| SQL_sort_order | 192 |

Table 16: Perturbation Types and Sample Counts

### A.10 Limitations of Rule-Based SQL Transpilers

Static analysis and syntax-based SQL transpiler (Rule-based transpiler) is an interesting direction for dialect SQL generation tasks. However, our observations highlight several limitations that make this approach less desirable compared to methods leveraging execution feedback.

#### Observation 1: Rule-Based Transpilers Still Require Execution Feedback

While tools like SQLGlot provide syntax-level SQL transpilation, they cannot guarantee semantic correctness or executable validity in the target dialect. As shown in Table 17, in many cases, SQLGlot generates syntactically valid but semantically incorrect queries, making execution feedback still necessary for validation and refinement.

#### Observation 2: Rule-Based Methods Can Hardly Do Multi-Round Refinement

Our method supports iterative refinement by injecting failing case inputs to the next round's prompt (similar to self-correction). However, rule-based transpilers like SQLGlot require manual updates from programming experts to improve over iterations, making them less adaptable in practice.

| Aspect | Content |
|---|---|
| SQLGlot Output | `SELECT T1.rating_score, T2.director_name FROM ratings AS T1 JOIN movies AS T2 ON T1.movie_id = T2.movie_id WHERE T2.movie_title = 'When Will I Be Loved'` |
| Execution Error | Error 1140 (42000): In aggregated query without GROUP BY, expression #2 of SELECT list contains nonaggregated column 'movie_platform.T2.director_name'; this is incompatible with sql_mode=only_full_group_by |
| Original SQLite Query | `SELECT T1.rating_score, T2.director_name FROM ratings AS T1 JOIN movies AS T2 ON T1.movie_id = T2.movie_id WHERE T2.movie_title = 'When Will I Be Loved'` |
| Correct MySQL Query | `SELECT avg(T1.rating_score) AS average_rating, T2.director_name FROM ratings AS T1 JOIN movies AS T2 ON T1.movie_id = T2.movie_id WHERE T2.movie_title = 'When Will I Be Loved' GROUP BY T2.director_name` |

Table 17: SQLGlot misses the required `GROUP BY` clause, which causes execution failure in MySQL under strict SQL modes.

#### Observation 3: Rule-Based Performance Is Worse

We compared the performance of SQLGlot against a single-round LLM-based translation.

Table 18 shows a clear gap in accuracy, further highlighting the limitations of relying solely on static transpilation.

#### Observation 4: Combine Usage of Rule-Based Transpiler and LLM Can Reduce LLM Call Cost

We also analyzed whether pre-filtering with SQLGlot can reduce the total number of LLM calls. Assuming SQLGlot correctly solves 35% of queries, and the LLM solves 56% of the remaining ones in

| Method | Accuracy (%) |
|---|---|
| LLM (API, 1 round) | 56.32 |
| SQLGlot | 35.18 |

Table 18: Accuracy Comparison: Rule-Based vs. LLM-Based Translation

each round (up to 3 rounds), the estimated number of API calls is shown in Table 19. Pre-filtering with SQLGlot results in roughly 35% savings in LLM calls, consistent with the success rate of SQLGlot. However, while this combination can reduce costs, it still necessitates the use of an LLM and does not overcome the fundamental limitations in semantic correctness and iterative refinement of purely rule-based approaches.

| Setting | LLM Calls | | |
|---|---|---|---|
| | 1k | 10k | 100k |
| LLM-only | 1,634 | 16,336 | 163,360 |
| SQLGlot + LLM | 1,062 | 10,618 | 106,184 |

Table 19: Estimated LLM API Calls with and without SQLGlot Pre-filtering for 3 rounds

### A.11 Execution feedback efficiency

To better understand the potential efficiency bottlenecks, we analyze execution time across databases of varying complexity using the MySQL engine. Specifically, we compare execution performance on the Spider dataset (with relatively small and simple databases) and the BIRD dataset (Li et al., 2023a) (which contains significantly larger and more complex schemas). The complexity of database can be shown in Table 20 through the "Avg. Rows per DB" metric.

We acknowledge that execution-based validation introduces overhead, but we argue that the cost remains acceptable, especially given the significant gains in data quality and model generalization. Furthermore, since execution feedback is only used during data generation (not inference), this cost is one-time and offline. Future improvements could involve caching, schema-aware pruning, or batched execution to further enhance scalability.

### A.12 Answer Extraction

Since LLM-based text-to-SQL generation often introduces variance, such as generating unrelated information or placing answers in undesired formats, we incorporate a regex-based answer extraction tool for robust evaluation. Common formatting issues include repeated questions, answers enclosed in code blocks (e.g., "`...`"), and additional explanations.

### A.13 Detailed Translation-based Bootstrapping Process

The original Spider dataset is based on SQLite SQL. We used GPT-4o API to generate MySQL and PostgreSQL SQL queries based on the given SQLite SQL, natural language questions, and table information (including table names and column names). The generation process followed a structured approach to ensure high accuracy and compatibility across SQL dialects. In 24, 25, we describe the prompt used for GPT-4o, which highlights key differences between SQLite SQL and PostgreSQL/Mysql SQL. The prompt also provides several input-output examples that illustrate how SQLite SQL should be transformed into the target SQL dialects. These examples help GPT-4o understand the conversion rules and adapt the syntax accordingly.

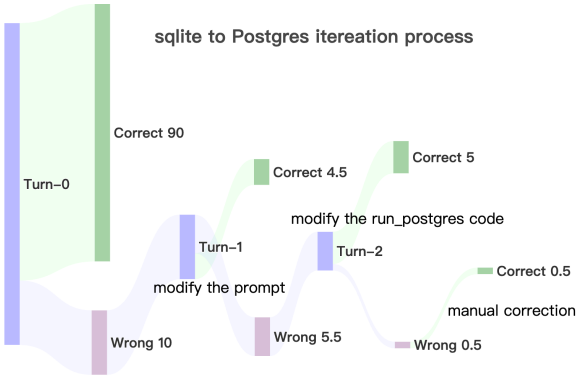


Figure 5: SQLite to PostgreSQL process.

**PostgreSQL Generation Process** Figure 5 illustrates the process of generating PostgreSQL SQL. In the first iteration of PostgreSQL generation, we found that around 680 queries failed with compilation errors. To address this, we enhanced the prompt by including additional PostgreSQL-specific features and updated the input-output examples with corrected versions of some failed queries from the first iteration (e.g., Ensure all 'JOIN' operations explicitly specify the 'ON' condition; When using GROUP BY, all selected non-aggregated columns must be explicitly listed in the GROUP BY clause). After applying the modified prompt, the number of incorrect queries decreased

| Dataset | Avg. Rows per DB | Avg. Import Time | Avg. Execution Time | Time for processing 8,000 Samples |
|---|---|---|---|---|
| Spider (small-scale) | 5,910.3 | 0.233 s | 0.00618 s | 87.2 s |
| BIRD (large-scale scenarios) | 256,231 | 21.647 s | 0.06149 s | 1,944 s |
| Training (7B model) | - | - | - | 11,100s |

Table 20: Execution Time Analysis (MySQL)

to about 400. Upon reviewing the errors, we discovered that most issues were caused by tables containing cells with improper formats (e.g., empty cells or invalid values like ""). To resolve this, we adjusted the code responsible for running PostgreSQL queries by skipping rows with problematic data during the conversion process. After implementing the data-cleaning step, only 30 queries remained incorrect. These were corrected manually to achieve a fully accurate PostgreSQL SQL dataset.

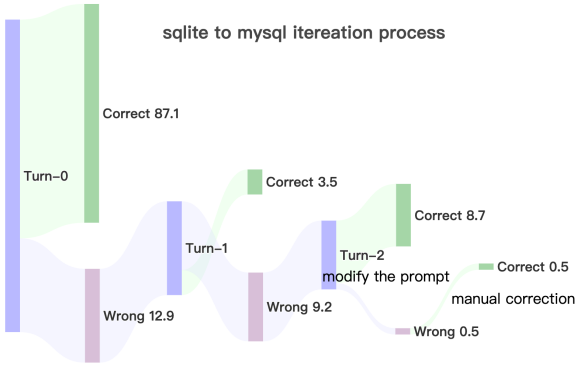


Figure 6: SQLite to MySQL process.

**Mysql Generation Process** As shown in Figure 6, the MySQL generation process followed a similar iterative approach. In the first iteration, GPT-4o generated MySQL SQL for all queries. After evaluation, we found that approximately 890 queries were incorrect due to compilation errors. Since the number of errors is still very high, we repeat the same iteration based on the incorrect samples. After the second iteration, there are still 630 examples that remain incorrect. To improve accuracy, we refined the prompt, adding more MySQL-specific features and incorporating corrected versions of some failed queries from the first two iterations. We asked GPT-4o to use backquotes for table names or column names when necessary. With the improved prompt, the number of incorrect queries dropped to around 20. Finally, we manually corrected these remaining 20 queries to achieve a fully accurate MySQL SQL dataset.

### A.14 Handling Complex SQL Features

Unlike static analysis or rule-based transpilers, our method leverages both SQL and question semantics. This allows it to better preserve and adapt complex functional behavior during translation. Below, Table 21 and Table 22 present examples demonstrating the capability of our method to correctly handle PostgreSQL-specific features, addressing the limitations often encountered by simpler translation approaches.

### A.15 Example of Execution-based Rejection Sampling

Table 23 presents an example of the proposed execution-based rejection sampling, where the question originates from the WikiSQL training set. We reformat the question to match the format used for our training and applied best-of-8 sampling with a cold-started model. In the upper section, we illustrate an example of generating new training data by sampling the correct Postgres SQL query. Among the eight generated queries, only one correctly queries the database using “date=January 16”, aligning with the question’s description. Although the other queries are semantically similar and syntactically correct, they fail to retrieve the correct answer.

The lower section provides an example of sampling for DPO training data. The question asks about “scored 24 points”, whereas the rejected answer retrieves values where the score is greater than 24, which is clearly incorrect.

| Example | SQLite Query | Translated PostgreSQL |
|---|---|---|
| Example 1 (No OVER in SQLite) | `SELECT T2.director_name FROM ratings AS T1 INNER JOIN movies AS T2 ON T1.movie_id = T2.movie_id WHERE T2.release_year BETWEEN 1960 AND 1985 GROUP BY T2.director_name HAVING COUNT(T2.movie_id) > 10 ORDER BY T2.director_name LIMIT 2` | `SELECT director_name, movie_title FROM (SELECT m.director_name, m.movie_title, m.movie_id, RANK() OVER (PARTITION BY m.director_id ORDER BY COUNT(r.rating_id) FILTER (WHERE r.rating_score::FLOAT > 5) DESC) as rank FROM movies AS m JOIN ratings r ON m.movie_id = r.movie_id WHERE m.release_year BETWEEN 1960 AND 1985 GROUP BY m.director_id, m.movie_id, m.director_name, m.movie_title) subquery WHERE rank::INTEGER = 1 AND director_name IN (SELECT director_name FROM movies WHERE release_year::INTEGER BETWEEN 1960 AND 1985 GROUP BY director_name HAVING COUNT(movie_id) >= 10) ORDER BY director_name LIMIT 2` |
| Example 2 (Window used) | `SELECT T3.years, T3.episode_id FROM (SELECT DISTINCT T2.years AS years, T2.episode_id, row_number() OVER (PARTITION BY (Window T2.episode_id ORDER BY T2.year) AS rn FROM person AS T1 INNER JOIN Award AS T2 ON T1.person_id = T2.person_id WHERE T2.award = used) 'Television' AND T2.award_category = 'Silver␣Gavel␣Award' AND T1.name = 'Constantine␣Makris' AND T2.result = 'Winner' AND T2.organization = 'American␣Bar␣Association␣Silver␣Gavel␣Awards␣for␣Media␣and␣the␣Arts') AS T3 GROUP BY T3.episode_id HAVING COUNT(T3.years - T3.rn) >= 2` | `SELECT T3.years, T3.episode_id FROM (SELECT DISTINCT T2.years AS years, T2.episode_id, row_number() OVER (PARTITION BY T2.episode_id ORDER BY T2.year) AS rn FROM person AS T1 JOIN Award AS T2 ON T1.person_id = T2.person_id WHERE T2.award = 'Television' AND T2.award_category = 'Silver Gavel␣Award' AND T1.name = 'Constantine␣Makris' AND T2.result = 'Winner' AND T2.organization = 'American␣Bar␣Association␣Silver␣Gavel␣Awards␣for␣Media␣and the␣Arts') AS T3 GROUP BY T3.episode_id, T3.years HAVING COUNT(T3.years - T3.rn::FLOAT) >= 2` |

Table 21: Translation Examples: PostgreSQL `OVER` (Window Function)

| Example | SQLite Query | Translated PostgreSQL |
|---|---|---|
| Example 3 (LIKE on date) | `SELECT T3.keyword_name FROM movie AS T1 INNER JOIN movie_keyword AS T2 ON T1.movie_id = T2.movie_id INNER JOIN keyword AS T3 ON T2.keyword_id = T3.keyword_id WHERE T1.release_date BETWEEN '2006-01-01' AND '2006-12-31' GROUP BY T3.keyword_name ORDER BY COUNT(T3.keyword_name) DESC LIMIT 1` | `SELECT T3.keyword_name FROM movie AS T1 JOIN movie_keyword AS T2 ON T1.movie_id = T2.movie_id JOIN keyword AS T3 ON T2.keyword_id = T3.keyword_id WHERE T1.release_date BETWEEN '2006-01-01' AND '2006-12-31' GROUP BY T3.keyword_name ORDER BY COUNT(T3.keyword_name) DESC LIMIT 1` |
| Example 4 (Regex-like keyword match) | `SELECT keyword_name FROM keyword WHERE keyword_name LIKE '%christmas%'` | `SELECT keyword_name FROM keyword WHERE keyword_name ILIKE '%christmas%'` |

Table 22: Translation Examples: PostgreSQL `ILIKE` (RegEx Like)

**Data Instance**

**Best of N**

[Input]:
You need to generate a Postgres SQL based on the following question and table information. Question: Who is the Opponent on January 16? Table and columns information: table_2_16946097_6: Date, H_A_N, Opponent, Score, Record.

[Outputs]:
SELECT Opponent FROM table_2_16946097_6 WHERE Date = 'January 16'
SELECT Opponent FROM table_2_16946097_6 WHERE Date = 'Jan 16'
SELECT Opponent FROM table_2_16946097_6 WHERE Date = 'Jan 16'
SELECT Opponent FROM table_2_16946097_6 WHERE Date = 'Jan 16'
SELECT Opponent FROM table_2_16946097_6 WHERE Date = 'Jan 16'
SELECT Opponent FROM table_2_16946097_6 WHERE Date = '1/16'
SELECT Opponent FROM table_2_16946097_6 WHERE Date = 'Jan 16'
SELECT Opponent FROM table_2_16946097_6 WHERE Date = '1/16'

[Correct Answer]:
SELECT Opponent FROM table_2_16946097_6 WHERE Date = 'January 16'

**Preference Pair**

[Input]:
You need to generate a Postgres SQL based on the following question and table information. Question: What was the record after the game in which the Hurricanes scored 24 points? Table and columns information: table_1_20928682_1: Game, Date, Opponent, Result, Hurricanes_points, Opponents, Record.

[Chosen Answer]:
SELECT table_1_20928682_1.Record FROM table_1_20928682_1
WHERE table_1_20928682_1.Hurricanes_points::FLOAT = 24

[Rejected Answer]:
SELECT table_1_20928682_1.Record FROM table_1_20928682_1
WHERE table_1_20928682_1.Hurricanes_points::FLOAT > 24

Table 23: Data instance of our iteration.

## Prompt Format for SQLite to PostgreSQL Conversion

**Prompt Description:**
You are an expert in SQL conversion. Convert SQLite SQL statements to PostgreSQL SQL while strictly following PostgreSQL's syntax.
**Important Instructions:**
1. **Input Format:** Each line in the input file follows this format:
SQLite SQL \t db_id
Example: `SELECT count(*) FROM head WHERE age > 56 department_management`
- "`SELECT count(*) FROM head WHERE age > 56`" is the SQLite SQL.
- "`department_management`" is the db_id.
2. **Output Format (STRICTLY ENFORCED):**
You must return the converted PostgreSQL SQL followed by the same db_id as input:
Example: `SELECT count(*) FROM head WHERE head.age::INTEGER > 56 department_management`
**Rules to Follow:**
- Do not add explanations, comments, or any extra text.
- Output must be one line per input, separated by a single '\t'.
- For column names, add the table name before each column in PostgreSQL SQL.
- Ensure db_id remains exactly as in the input.
- Ensure explicit column references (e.g., `table.column`).
- When using GROUP BY, all selected non-aggregated columns must be explicitly listed in the GROUP BY clause to avoid errors in PostgreSQL.
- Ensure all 'JOIN' operations explicitly specify the 'ON' condition. Avoid using implicit joins or missing 'ON' conditions, as PostgreSQL requires explicitly defined relationships between tables.
- If a table has an alias in the 'FROM' or 'JOIN' clause, always use the alias instead of the original table name in 'SELECT', 'WHERE', and other clauses.
- for SELECT DISTINCT, ORDER BY expressions must appear in select list
- Ensure that tables are referenced in 'JOIN' statements in the correct order: a table must be defined before being used in an 'ON' condition.

**Example 1:**
**Input:**
`SELECT DISTINCT T1.player_name, T1.birthday FROM Player AS T1 JOIN Player_Attributes AS T2 ON T1.player_api_id = T2.player_api_id ORDER BY potential DESC LIMIT 5 soccer_1`
**Output:**
`SELECT DISTINCT T1.player_name, T1.birthday, T2.potential::FLOAT FROM Player AS T1 JOIN Player_Attributes AS T2 ON T1.player_api_id = T2.player_api_id ORDER BY T2.potential::FLOAT DESC LIMIT 5 soccer_1`

**Example 2:**
**Input:**
SELECT count(*) FROM head WHERE age > 56 `department_management`
**Output:**
SELECT count(*) FROM head WHERE head.age::INTEGER > 56 `department_management`

**Example 3:**
**Input:**
SELECT avg(lat), avg(long) FROM station WHERE city = "San Jose" `bike_1`
**Output:**
SELECT AVG(lat::FLOAT), AVG(long::FLOAT) FROM station WHERE station.city = "San Jose" `bike_1`

**Example 4:**
**Input:**
SELECT T1.age FROM Person AS T1 JOIN PersonFriend AS T2 ON T1.name = T2.friend WHERE T2.name = 'Zach' AND T2.year = (SELECT max(YEAR) FROM PersonFriend WHERE name = 'Zach') `network_2`
**Output:**
SELECT T1.age FROM Person AS T1 JOIN PersonFriend AS T2 ON T1.name = T2.friend WHERE T2.name = 'Zach' AND T2.year::FLOAT = (SELECT MAX(YEAR::FLOAT) FROM PersonFriend WHERE name = 'Zach') `network_2`

**Now, convert the following SQLite SQL to PostgreSQL SQL. Output strictly in format: SQL \t db_id.**

Table 24: Prompt example for converting SQLite SQL to PostgreSQL SQL.

## Prompt Format for SQLite to MySQL Conversion

**Prompt Description:**
You are an expert in SQL conversion. Convert SQLite SQL statements to MySQL SQL while strictly following MySQL's syntax.
**Important Instructions:**
1. **Input Format:** Each line in the input file follows this format:
Index \t SQLite SQL \t db_id
Example: `1 SELECT T3.course_name , count(*) FROM students AS T1 JOIN student_course_registrations AS T2 ON T1.student_id = T2.student_id JOIN courses AS T3 ON T2.course_id = T3.course_id GROUP BY T2.course_id student_assessment`
2. **Output Format (STRICTLY ENFORCED):**
Just output the converted MySQL SQL query (do not include index or db_id).
Example: `SELECT T3.course_name , count(*) FROM Students AS T1 JOIN Student_Course_Registrations AS T2 ON T1.student_id = T2.student_id JOIN Courses AS T3 ON T2.course_id = T3.course_id GROUP BY T2.course_id, T3.course_name`
**Rules to Follow:**
- Do not add explanations, comments, or any extra text.
- Output must be one line per input, separated by a single '\t'.
- When using `GROUP BY`, all selected non-aggregated columns or tables must be explicitly listed in the `GROUP BY` clause to avoid errors in MySQL.
- When writing table names in MySQL, case matters. Refer to the provided table information to ensure correct casing.
- Use backquotes for table name or column name when necessary.
- This version of MySQL doesn't yet support 'LIMIT & IN/ALL/ANY/SOME subquery'

**Example 1:**
**Input:**
2 `SELECT T1.campus , sum(T2.degrees) FROM campuses AS T1 JOIN degrees AS T2 ON T1.id = T2.campus WHERE T2.year >= 1998 AND T2.year <= 2002 GROUP BY T1.campus csu_1`
**Output:**
`SELECT T1.campus, SUM(T2.degrees) FROM Campuses AS T1 JOIN degrees AS T2 ON T1.id = T2.campus WHERE T2.year >= 1998 AND T2.year <= 2002 GROUP BY T1.campus`

**Example 2:**
**Input:**
3 `SELECT T1.faculty, avg(T2.salary) FROM faculties AS T1 JOIN salaries AS T2 ON T1.faculty_id = T2.faculty_id GROUP BY T1.faculty university_pay`
**Output:**
`SELECT T1.faculty, AVG(T2.salary) FROM Faculties AS T1 JOIN Salaries AS T2 ON T1.faculty_id = T2.faculty_id GROUP BY T1.faculty`

**Now, convert the following SQLite SQL to MySQL SQL. Output strictly in the format described above.**

Table 25: Prompt example for converting SQLite SQL to MySQL SQL.