NL To Performant SQL

Columbia COMS4705 Final Project Report

Keywords: *NL2SQL, SQL efficiency, Direct Preference Optimization, Text-to-SQL, model fine-tuning, performance-aware SQL*

Elaine Ang
Department of Computer Science
Columbia University
ra3448@columbia.edu

Rohith Kanathur
Department of Computer Science
Columbia University
rk3443@columbia.edu

Vivian Lu
Department of Statistics
Columbia University
jl7244@columbia.edu

Abstract

While current Natural Language to SQL (NL-to-SQL) systems have achieved significant strides in logical correctness, they frequently overlook query execution performance—a critical factor for production deployment. This project investigates the potential of preference-based alignment strategies to optimize for computational efficiency alongside semantic accuracy. Using Qwen/Qwen3-4B-Instruct as our experimental baseline, we employ a two-stage alignment strategy to drive high-performance SQL generation. First, we apply Direct Preference Optimization (DPO) using a manually curated dataset of 287 semantically equivalent query pairs, explicitly distinguishing between efficient ("fast") and inefficient ("slow") designed to avoid specific anti-patterns. Second, we utilize execution-aware Reinforcement Learning (RL) on the BIRD training set, leveraging a composite reward function that balances accuracy with latency reduction. Preliminary results indicate a 7% improvement over baseline in the Relative Valid Efficiency Score (R-VES) on the BIRD Mini-Dev evaluation set.

1 Key Information to include

- TA mentor: Joey Huang
- External collaborators (if no, indicate “No”): No
- Sharing project (if no, indicate “No”): No

2 Introduction

Recent advancements in Large Language Models (LLMs) have fundamentally altered the landscape of Natural Language to SQL (NL-to-SQL) generation. By leveraging Supervised Fine-Tuning (SFT) and In-Context Learning (ICL), modern systems can now translate natural language questions into valid SQL queries with high semantic accuracy. However, this focus on logical correctness often masks a critical deficiency: execution performance. In most production environments, a query that yields the correct result but consumes excessive resources or incurs high latency is functionally equivalent to a failure. As databases grow in scale and complexity, the ability to generate SQL that is not merely correct, but also computationally efficient, becomes a primary barrier to real-world deployment.

Generating high-performance SQL is significantly harder than ensuring syntactic correctness because it requires navigating the opaque heuristics of database query planners. Often, the most intuitively

readable SQL—such as using `OR` conditions for filtering or Common Table Expressions (CTEs) for modularity—can inadvertently trigger inefficient execution plans. For instance, in many PostgreSQL configurations, a CTE might be materialized fully before filtering, causing massive memory overhead, whereas a less readable nested subquery would allow the planner to "push down" predicates and filter data early. Similarly, an `OR` clause might force a full table scan, while a syntactically longer `UNION ALL` structure would trigger efficient index scans. Naive LLMs, trained on vast corpora of human-written code, frequently mimic these readable but inefficient "anti-patterns", failing to account for the specific "quirks" of the underlying query optimizer.

Contribution To address this, we propose a methodology that specifically aligns the model to avoid these planner pitfalls. Unlike previous methods that rely on external re-rankers to guess the best query, we aim to internalize the logic of an expert database administrator directly into the model's weights. A cornerstone of our approach is the creation of a specialized preference dataset [1] (available at octxmas/text-to-sql-performance). By generalizing patterns from a wide range of PostgreSQL community discussions and optimization guides, we manually curated 287 pairs of semantically equivalent queries. These pairs explicitly contrast "bad" formulations—those that trigger planner quirks like unnecessary materialization or poor index usage—against "good" formulations that coax the planner into optimal execution paths (e.g. preferring `NOT EXISTS` over `NOT IN`, ensuring pre-aggregating data before joins, etc.).

We utilize this dataset in a Direct Preference Optimization (DPO) stage, teaching the model to instinctively prefer these efficient structures. We then reinforce this behavior through execution-aware Reinforcement Learning (RL) on the BIRD benchmark. By employing a composite reward function that rewards correctness while penalizing latency, we enable the model to further refine its generation strategy against real-world execution feedback.

We evaluate this methodology using Qwen/Qwen3-4B-Instruct[2] as our base model. Our experiments demonstrate that the combination of static preference learning (DPO) and dynamic feedback (RL) leads to 7% improvement over baseline in the Relative Valid Efficiency Score (R-VES) on BIRD Mini-Dev dataset. However, detailed analysis reveals that these gains are primarily driven by improved execution accuracy rather than pure latency reduction, likely due to the limited potentials for structural variety within our evaluation dataset. Nonetheless, our study shows that execution-centric alignment can drive measurable improvements of a compact small model. All relevant source code and qualitative results for analysis are available at `https://github.com/ElaineAng/nlp-final`.

Remainder of this report is organized as follows: In Section 3, we discuss related work in NL-to-SQL generation, execution-aware alignment, and preference learning. Section 4 details our methodology, including dataset construction, DPO training, and RL reward design. Section 5 presents our experimental setup and results. Section 6 does a qualitative analysis of the result, and discusses limitations and future directions. Finally, Section 7 presents an overall conclusion of this study.

3 Related Work

Semantic Foundations and Limitations The field of NL-to-SQL has undergone a significant transformation, driven largely by the capabilities of LLMs employing Supervised Fine-Tuning (SFT) and In-Context Learning (ICL). Early research primarily prioritized maximizing semantic accuracy, a paradigm established by the Spider benchmark [3]. Spider defined a cross-domain semantic parsing task comprising 10,181 questions and 5,693 unique SQL queries across 200 databases, focusing on generalization to unseen schemas. The primary metrics, Exact Match (EM) and Execution Accuracy (EX), incentivized models to produce logically correct SQL. However, subsequent analyses have revealed the limitations of this correctness-first focus. Spider's schemas are often simpler than those in enterprise environments [4], and LLM-based systems exhibit brittleness, with performance dropping by over 10% when facing paraphrased questions [5]. These findings suggest that high EX scores on standard benchmarks do not necessarily translate to the structural robustness required for real-world deployment.

The "Efficiency Imperative" Beyond semantic correctness, a critical gap remains regarding query execution performance. For practical adoption, LLM-generated SQL must be optimized for latency, join efficiency, and index utilization [6, 7, 8, 9]. This "efficiency imperative" motivated the BIRD benchmark [10], which introduced 12,751 question–SQL pairs across 95 massive databases (33.4 GB). Crucially, BIRD introduced the Relative Valid Efficiency Score (R-VES), which jointly evaluates

execution accuracy and efficiency. This metric reframes NL-to-SQL as a multi-objective optimization problem, requiring methods that improve runtime performance without degrading semantic fidelity.

Alignment Strategies: DPO and RL To address these multi-objective challenges, researchers have turned to execution-aware alignment methods. In the realm of static preference learning, Direct Preference Optimization (DPO) [11] has emerged as a powerful algorithm for optimizing policies using binary preference data. Its efficacy in structured tasks is supported by extensions like Focused-DPO [12], which targets error-prone code regions, and synthetic data strategies that construct DPO datasets by contrasting outputs from large and small models [13].

Complementing static alignment, dynamic approaches incorporate real-time performance signals via Reinforcement Learning (RL). Early work demonstrated that selecting prompts based on latency could improve system throughput [7]. More recent frameworks leverage active database interaction: ReEx-SQL [14] utilizes RL to iteratively refine queries based on intermediate execution feedback, while HES-SQL [15] explicitly optimizes query latency through composite rewards addressing both accuracy and efficiency. By integrating self-distillation with RL, HES-SQL demonstrated efficiency gains of 11% to 20% over supervised baselines.

Building on these advancements in execution-aware alignment, our work investigates the extent to which efficiency can be optimized strictly within the constraints of a compact architecture. We focus on a 4B parameter model, exploring whether a specialized alignment strategy—applying DPO to preference pairs based on query structures (fast vs. slow) followed by RL with a composite latency-aware reward function—is sufficient to drive efficiency gains without compromising semantic accuracy.

4 Approach

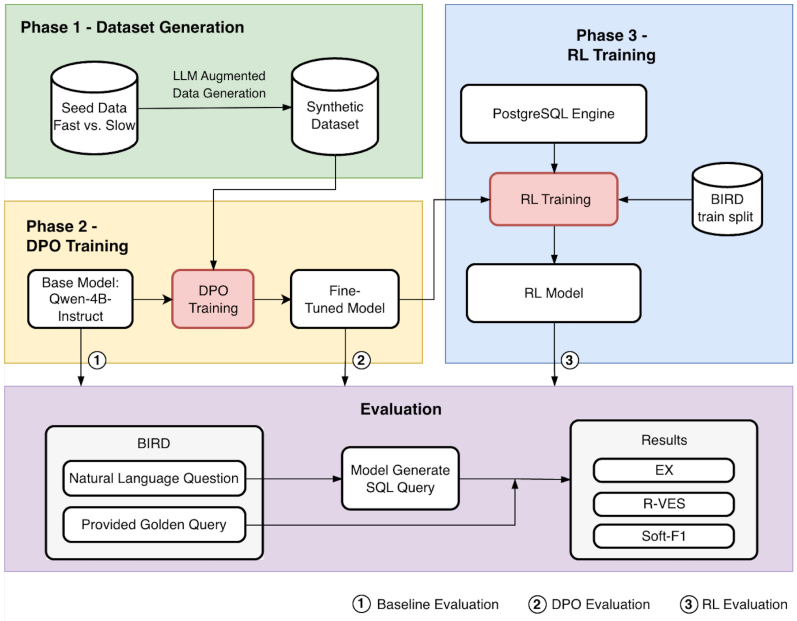

Figure 1: Approach Overview

Figure 1 illustrates our overall approach. Using `Qwen/Qwen3-4B-Instruct-2507` [2] as our base model, we first fine-tune it using Direct Preference Optimization (DPO) on a manually constructed and LLM-augmented synthetic dataset consisting of efficient/inefficient SQL query pairs, and then apply RL on the *train split* from the BIRD benchmark [16]. All models are evaluated on the BIRD Mini-Dev set (500 examples) to measure execution accuracy and performance. We compare the baseline stats, the DPO-only results, and the DPO+RL results to assess whether performance improvements come without degrading execution accuracy.

We use standard DPO loss for the DPO training, with detailed hyperparameters listed in section 5.3. For RL training, we use Group Relative Policy Optimization (GRPO) with a composite reward that balances three objectives: semantic correctness, schema awareness, and execution efficiency. The reward calculation operates in two phases. It first assigns base rewards: correctly executing queries receive the highest reward ($W_{correct}$), while queries demonstrating proper schema linking receive a bonus (W_{link}) even if execution fails, and syntactically invalid queries are penalized (W_{syntax}). Then, among queries that produce correct results, an efficiency bonus is applied using normalized execution latency, rewarding faster queries while penalizing slower ones. This design encourages the model to generate SQL queries that are not only semantically correct but also computationally efficient. The complete reward design is shown in Algorithm 1.

Algorithm 1 GRPO Reward Design

Input: Query group $Q = \{q_1, \ldots, q_n\}$, gold result R_g, gold schema S_g, postgres connection D
Output: Reward vector $\mathbf{r} \in \mathbb{R}^n$

Constants: $W_{correct} = 3.0$, $W_{link} = 1.5$, $W_{syntax} = -1.0$, $W_{fail} = -0.5$, $\lambda_e = 1$

// Rewards and latencies initialization
Initialize $\mathbf{r} \leftarrow \mathbf{0}_n$, $\mathbf{c} \leftarrow \mathbf{0}_n$

// Phase 1: Evaluate queries and assign base rewards
for $i = 1$ **to** n **do**
  $linked \leftarrow$ CheckSchemaLink$(Q[i], S_g)$

  if not IsValidSyntax$(Q[i])$ **then**
    $r_i \leftarrow W_{syntax}$
  else
    try:
      $(R_i, c_i) \leftarrow$ ExecuteAndMeasure$(Q[i], D)$
      if Equals(R_i, R_g) **then** $r_i \leftarrow W_{correct}$
      else $r_i \leftarrow W_{fail} + (W_{link}$ **if** $linked$ **else** $0)$
    catch:
      $r_i \leftarrow W_{fail} + (W_{link}$ **if** $linked$ **else** $0)$
  end if
end for

// Phase 2: Add efficiency bonus to correct queries
$I \leftarrow \{i \mid r_i = W_{correct}\}$
if $|I| > 1$ **then**
  Compute μ, σ from latencies $\{c_i \mid i \in I\}$
  for each $i \in I$ **do**
    $r_i \leftarrow r_i + \lambda_e \cdot \tanh\left(\frac{\mu - c_i}{\sigma + \epsilon}\right)$
  end for
end if
return $\mathbf{r}$

5 Experiments

5.1 Data

Since none of the existing benchmarks provided reference queries with performance context, we created a synthetic dataset of efficient/inefficient SQL query pairs to align our models towards using a more performant SQL structure. We hand-crafted 26 seed examples derived from PostgreSQL performance tuning blogs [17], then used these examples to prompt Claude Sonnet 4.5 to generate a larger corpus of 287 semantically equivalent query variations. We published the resulting dataset on Hugging Face [1]. In addition to the synthetic dataset, we incorporated the BIRD benchmark [16] into our training and evaluation pipeline. We used the official BIRD Train split, consisting of 9,428 examples across 69 databases, for reinforcement learning with GRPO. We evaluated our model on

the BIRD Mini-Dev split (500 examples) to measure performance on held-out data from the same distribution.

One side note is that we also performed the same set of evaluations on the Spider 1.0 dataset to measure semantic accuracy and generalization, and we get a baseline execution accuracy of 69.6%. We omitted details related to the Spider 1.0 evaluation here, since we didn't see any significant changes compare to the base model. It's likely due to Spider 1.0 dataset being overly simplistic and doesn't have enough performance variations for the model to demonstrate a difference.

5.2 Evaluation method

We evaluated both the baseline and fine-tuned models using identical procedures. We wrote a simple inference script that takes a specified model, a natural language question, json-based database schema, and a natural language hint as input, and does a one-shot prediction of SQL queries with temperature of 0 for the BIRD Mini-Dev dataset. We then used the standard evaluation scripts provided by the original benchmark authors [18] to compute execution accuracy and efficiency metrics.

The evaluation is broken down into 3 scores: Execution Accuracy (EX) measures the proportion of examples for which the executed results of the predicted and ground-truth SQL queries are identical. R-VES evaluates the efficiency of valid SQL queries i.e., queries that return results consistent with the ground-truth. The soft F1-score metric measures "partial" correctness by determining the similarity between the tables produced by predicted queries and those of the ground truth, distinguishing between "completely wrong hallucination" and "nearly correct logic." [16].

5.3 Experimental details

We ran all inference the experiments on a Google Cloud Compute instance equipped with a single NVIDIA L4 GPU (16GB VRAM) and 4 vCPUs. To accommodate memory constraints, we employed 16-bit quantization for model weights. We performed database query evaluation on a MacBook Pro with an M4 Pro chip and 24GB of memory, running PostgreSQL 17 (stable release). We configured PostgreSQL with `-max_connections=300` while maintaining default settings for all other parameters. Inference over the BIRD dev set (500 examples) took $\approx$ 40 minutes.

For training, we ran everything on Tinker using the hyperparameters listed in table 1 and 2 below.

Learning Rate	Batch Size	Epochs	DPO Beta
1×10^{-5}	32	1	0.1

Table 1: DPO Training Hyperparameters

Learning Rate	Batch Size	Group Size	Temperature
1×10^{-6}	32	8	0.7

Table 2: GRPO Training Hyperparameters

DPO finished in less than 10 minutes with the relatively small synthetic dataset. The training progressed smoothly, with the accuracy improving from 0.5 at the start to 0.9 when we finish, and the loss converges from 0.69 to 0.59.

We intentionally used a smaller learning rate for GRPO training to prevent DPO forgetting and minimize reward noise. Temperature is set to 0.7 to improve diversity of generated query. Group size is set to 8 to ensure each group has enough exploration to get a diverse reward, while keeping the training time reasonable. Using a batch size of 32 allows us to balance training speed and gradient quality. With this setup, on average it took about 15 minutes per batch, and we trained for a total of 74 steps (around 18 hours). This only exhausts about 25% of the entire BIRD training set. The mean reward for each batch keeps oscillating around 1.8 throughout the training, with a high of 2.3 and a low of 1.5, likely due to the variance in the difficulty of each batch. Unfortunately due to time constraints we were not able to run more training steps to see if the reward could stabilize or improve further. We keeps tracking the batch with the highest mean reward and execution accuracy, and picked the best batch checkpoint (batch 66) for final evaluation.

5.4 Results

We provide the baseline, DPO, and GRPO results below for EX, R-VES, and Soft-F1, separated by different difficulty levels.

Metric	Simple (148)	Moderate (250)	Challenging (102)	Total (500)
Baseline	26.35	18.00	13.73	19.60
DPO	**27.03**	18.40	15.69	20.40
GRPO	26.35	**19.60**	**16.67**	**21.00**

Table 3: BIRD EX Score Comparison

Metric	Simple (148)	Moderate (250)	Challenging (102)	Total (500)
Baseline	26.42	17.82	13.73	19.53
DPO	**26.93**	18.28	15.69	20.31
GRPO	26.42	**19.46**	**16.54**	**20.92**

Table 4: BIRD R-VES Score Comparison

Metric	Simple (148)	Moderate (250)	Challenging (102)	Total (500)
Baseline	26.16	18.90	13.10	19.86
DPO	**26.84**	18.77	14.95	20.38
GRPO	25.21	**20.57**	**16.58**	**21.13**

Table 5: BIRD Soft-F1 Score Comparison

Since R-VES involves getting runtime stats from postgres execution, we ran the evaluation multiple times to reduce noise and reported the highest scores here, which is consistent with the official BIRD evaluation process.

We manually experimented with multiple hyperparameters for both DPO and GRPO training, but couldn't see a significant difference in the final evaluation results for the various combinations. During GRPO training, we observed that correctly executed queries tends to have low variance in execution latency, which makes the latency rewards noisy by picking up minor fluctuations. This might be due to either the base model limitations in generating diverse query structures, or the BIRD training set not having enough high-contrast performance examples.

Overall, we see a small but consistent improvement across all difficulty levels with our fine-tuned model. The improvement is most significant in the "Challenging" category with RL training, demonstrating the effectiveness of our reward design in improving overall performance on more complex query tasks. We break down the number of executable queries and correct queries (which is a subset of executable queries) by difficulty levels in Table 6.

It's worth noting that having the same number of correct queries does not mean the correctly executed queries are the same. As we'll show in the next section, there are both improvements and regressions for our fine tuned models for query generation, and the improvements seem to be mostly attributed to execution accuracy instead of pure latency reduction.

6 Analysis

We logged all relevant metrics during evaluation for post-hoc analysis to better understand the improvements and regressions, and we present a couple interesting examples below.

Varied Structures We observed that our fine-tuned models give a different structure from the base model for certain queries, picking up the query structure preference in our synthetic DPO training dataset. For example, for the question *"How many members attended the "Women's Soccer" event?"*, results from the base model and the fine-tuned models (which all executed correctly) are shown in listing 1

	Simple (148)		Moderate (250)		Challenging (102)		Total (500)	
Model	**Exec**	**Correct**	**Exec**	**Correct**	**Exec**	**Correct**	**Exec**	**Correct**
Baseline	58	36	89	39	34	12	181	87
DPO	61	36	86	41	38	15	185	92
GRPO	58	34	**92**	**42**	**40**	**16**	**190**	**92**

Table 6: Breakdown of executable and correct queries by difficulty level

Listing 1: Different SQL Structures for the same question

```
-- Base Model's output (correct execution)
SELECT COUNT(*) FROM Attendance
WHERE Attendance.link_to_event IN (
    SELECT event_id FROM Event WHERE event_name = 'Women''s Soccer'
);

-- DPO and GRPO Models' output (correct execution)
SELECT COUNT(*) FROM Attendance
JOIN Event ON Attendance.link_to_event = Event.event_id
WHERE Event.event_name = 'Women''s Soccer';
```

The base model uses subqueries in the `WHERE` clause with an `IN` condition; whereas both DPO and GRPO models uses a `JOIN` structure instead. Exactly which one is faster may depend on the query optimizer, data distribution and indexing, but generally `JOIN` is a safer choice over `IN` on subqueries [19].

Attempting to process less data Another interesting observation is that the fine-tuned model seems to be attempting to get better performance by reducing the amount of data to process, which is the right "intuition", but leads to a mix of improvements and regressions. As we show in listing 2, for the question *"Write the full name of the member who spent money for water, veggie tray and supplies and include the cost of it"*, the base model selects a potentially large `expense_description` column from the "Expense" table, while the fine-tuned model only selects the necessary name and cost columns. For this specific question, that choice leads to not just faster, but also correct execution since the original question isn't asking for a description at all.

Listing 2: Selecting fewer columns for performance which also improves correctness

```
-- Base Model's output (failed since the expense_description column is not needed)
SELECT m.first_name, m.last_name, e.expense_description, e.cost
FROM Expense e
JOIN Member m ON e.link_to_member = m.member_id
WHERE e.expense_description = 'Water, Veggie tray, supplies';

-- DPO and GRPO Models' output (correct execution)
SELECT m.first_name, m.last_name, e.cost
FROM Expense e
JOIN Member m ON e.link_to_member = m.member_id
WHERE e.expense_description = 'Water, Veggie tray, supplies';
```

We see similar cases where the fine-tuned model adds `DISTINCT` to eliminates duplicates, potentially trying to be faster, for questions that implicitly require unique results. The base model didn't remove duplicates, leading to incorrect answers, while the fine-tuned models got it right.

However, in other cases, optimizing for less data processing leads to correctness issues. As shown in listing 3, for the question *"Which event has the lowest cost?"*, The fine-tuned models omitted a necessary `JOIN` to the "Budget" table, which leads to failed execution since the `Expense` table doesn't have a `link_to_event` column.

Listing 3: Omitting necessary JOIN for performance which degrades correctness

```
-- Base Model's output (correct execution)
SELECT e.event_name FROM Event e
```

```
JOIN Budget b ON e.event_id = b.link_to_event
JOIN Expense ex ON b.budget_id = ex.link_to_budget
ORDER BY ex.cost ASC LIMIT 1;

-- DPO and GRPO's output (failed since 'link_to_event' doesn't exist in Expense)
SELECT e.event_name FROM Event e
JOIN Expense exp ON e.event_id = exp.link_to_event
ORDER BY exp.cost ASC LIMIT 1;
```

We see similar regressions like this where the fine-tuned model tries to use smaller tables to reduce data processing, but the used table doesn't have the necessary information to answer the question, leading to incorrect results. We suspect these correctness regressions over baseline should go away with more RL training steps, since our reward design still prioritizes correctness over performance.

Limitations We discuss some limitations with this study, and directions for future work. First, our manually curated preference dataset consists of only 287 query pairs, a relatively small sample that may not capture the full breadth of possible SQL optimization patterns. Furthermore, the optimization heuristics in this dataset are derived primarily from PostgreSQL behaviors. While many principles (like early filtering) are universal, specific quirks—such as CTE materialization policies—may vary across database engines like SQLite or MySQL, potentially limiting the cross-dialect generalization of our learned policies. Collecting high quality and diverse preference data that covers a wider range of databases and optimization strategies remains an open challenge.

Second, our base model (Qwen3-4B-Instruct) already demonstrates relatively strong performance on the BIRD benchmark. Using an instruction-tuned model as our starting point may have constrained the potential gains from further alignment, as the model had already internalized many general NL-to-SQL patterns. Future work could explore starting from a pre-trained LLM without any post-training to better isolate the effects of our alignment strategies.

Lastly, our RL and evaluation phase revealed significant shortcomings in existing benchmark data for performance tuning. We observed that the BIRD training set often lacks the distinct performance tiers necessary to provide a strong learning signal; many generated query variations either execute within similar millisecond-level windows, or doesn't have meaningful structural differences. Consequently, the latency-based reward function frequently struggled to distinguish meaningful optimization from system noise, leading to oscillating mean rewards during training. Similarly, the BIRD Mini-Dev evaluation set lacks queries which has potentials for significant performance variation. This highlights a scarcity of high-quality, large-scale datasets designed specifically for execution performance. Generating synthetic environments with high-contrast performance signals—particularly for specialized domains like Graph Databases where latency is a primary objective—represents a valuable and necessary direction for future research.

7 Conclusion

The primary objective of this study was to improve query execution speed by aligning Large Language Models to avoid anti-patterns and generate "optimizer-friendly" SQL structures. By employing Direct Preference Optimization (DPO) on a curated dataset of efficient versus inefficient query pairs, we aimed to internalize expert database heuristics directly into the model's weights.

Our approach yielded a measurable improvement on the BIRD benchmark, achieving a 7% increase in the Relative Valid Efficiency Score (R-VES), improving from a baseline of 19.53 to 20.92. The improvement was most pronounced in the "Challenging" difficulty category, where the R-VES rose from 13.73 to 16.54. However, a granular analysis of these results reveals that the efficiency score improvements were primarily driven by a higher rate of execution correctness rather than pure latency reduction.

While we initially aimed for speed, the evaluation suggests that optimizing for efficient structures may have had the side effect of improving semantic accuracy, likely by encouraging the model to generate queries that process the minimal amount of data necessary for the task. Qualitatively, the fine-tuned model demonstrates evidence of learning to adopt better practices, such as substituting inefficient subqueries with JOIN operations. However, we could not isolate a measurable reduction in

execution latency for valid queries, likely due to the limited size of the evaluation dataset and the lack of high-contrast performance tiers in the BIRD Mini-Dev set.

This study demonstrates that while performance-aware alignment can significantly boost overall benchmark scores and encourage better coding standards, quantifying pure efficiency gains requires larger, more specialized datasets designed to isolate execution time from logical correctness.

8 Team Contributions

Team Members

Elaine Ang: Led project ideation and execution, designed the overall approach, constructed the DPO dataset, designed and implemented evaluation strategies, developed and ran the final RL training, performed evaluations and analysis, and wrote significant portions of the project reports.

Rohith Kanathur: Responsible for coding and fine-tuning the base model with DPO, implementing the 1st iteration/skeleton of RL code, performing evaluation, qualitative analysis, developing data setup and benchmarking scripts, and contributing to the project reports.

Vivian Lu: Contributed to project discussions and proposal development, supported the design of our evaluation approach, analyzed model outputs across benchmarks, and assisted with writing and organizing the final report.

Honorable Mentions:

Claude Sonnet 4.5: Assisted in expanding the initial set of efficient/inefficient SQL query pairs.

Antigravity/Claude Code: Relentlessly iterated on the implementation; drafting and revising the scripts for inferencing, training, evaluation, and analysis.

References

[1] Elaine Ang. Synthetic dataset for efficient and inefficient sql queries, Oct 2025. URL https://huggingface.co/datasets/octxmas/text-to-sql-performance.

[2] Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

[3] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018. URL https://aclanthology.org/D18-1425.pdf.

[4] Yi Zhang, Jan Deriu, George Katsogiannis-Meimarakis, Catherine Kosten, Georgia Koutrika, and Kurt Stockinger. Sciencebenchmark: A complex real-world benchmark for evaluating natural language to sql systems, 2023. URL https://arxiv.org/abs/2306.04743.

[5] Mohammadtaher Safarzadeh, Afshin Oroojlooyjadid, and Dan Roth. Evaluating nl2sql via sql2nl. *arXiv*, 2025. URL https://arxiv.org/abs/2509.04657.

[6] Xiaozheng Du, Shijing Hu, Feng Zhou, Cheng Wang, and Binh Minh Nguyen. Fi-nl2py2sql: Financial industry nl2sql innovation model based on python and large language model. *Future Internet*, 17(1), 2025. ISSN 1999-5903. doi: 10.3390/fi17010012. URL https://www.mdpi.com/1999-5903/17/1/12.

[7] Sairam Gurajada, Eser Kandogan, and Sajjadur Rahman. Effectiveness of prompt optimization in nl2sql systems. *arXiv.org*, 2025.

[8] Kapil Vaidya, Jialin Ding, Sebastian Kosak, D. Kernert, Chuan Lei, Xiao Qin, Abhinav Tripathy, Ramesh Balan, B. Narayanaswamy, and Tim Kraska. Tailorsql: An nl2sql system tailored to your query workload. *arXiv.org*, 2025.

[9] Yuyang Song, Hanxu Yan, Jiale Lao, Yibo Wang, Yufei Li, Yuanchun Zhou, Jianguo Wang, and Mingjie Tang. Quite: A query rewrite system beyond rules with llm agents. *arXiv.org*, 2025.

[10] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 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 (NeurIPS)*, 36, 2024. URL https://bird-bench.github.io/.

[11] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: your language model is secretly a reward model. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[12] Zhen Yang, Ming Wang, Hao Chen, and Jie Li. Focused-dpo: Enhancing code generation through focused preference optimization. In *arXiv preprint arXiv:2502.11475*, 2025. URL https://arxiv.org/html/2502.11475v1.

[13] Jiaxi Yang, Binyuan Hui, Min Yang, Jian Yang, Junyang Lin, and Chang Zhou. Synthesizing text-to-sql data from weak and strong llms. *arXiv*, 2024. URL https://arxiv.org/abs/2408.03256.

[14] Yaxun Dai, Wenxuan Xie, Xialie Zhuang, Tianyu Yang, Yiying Yang, Haiqin Yang, Yuhang Zhao, Pingfu Chao, and Wenhao Jiang. Reex-sql: Reasoning with execution-aware reinforcement learning for text-to-sql. *arXiv*, 2025. URL https://arxiv.org/abs/2505.12768.

[15] Suming Qiu, Jing Li, Zhicheng Zhou, Junjie Huang, Linyuan Qiu, and Zhijie Sun. Hes-sql: Hybrid reasoning for efficient text-to-sql with structural skeleton guidance. *arXiv.org*, 2025.

[16] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 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, 2024.

[17] Lukas Fittl. pganalyze blog, Nov 2025. URL https://pganalyze.com/blog.

[18] Bird Benchmark Team. Bird evaluation scripts, Aug 2025. URL https://github.com/bird-bench/mini_dev/tree/main/evaluation.

[19] Harold Giménez. Postgresql performance considerations, Jan 2025. URL https://thoughtbot.com/blog/postgresql-performance-considerations#use-more-joins.