ReGuard: Regulating Query Rewriting for Utility-Aware Differentially Private Text-to-SQL

Anonymous ACL submission

Abstract

Text-to-SQL enables natural language access to databases and commonly relies on query rewriting to resolve ambiguity in user queries. As such systems increasingly operate over sensitive data, protecting query privacy has become a practical concern. A natural approach is to apply differential privacy (DP) on top of existing systems by adding noise to query results. However, under DP constraints, making queries clearer through rewriting inadvertently increases the effective sensitivity of query results, thereby requiring stronger noise addition and negating utility gains. To address this challenge, we propose ReGuard, a utility-aware Text-to-DP-SQL system that regulates query rewriting and jointly optimizes rewriting and DP-based query answering. ReGuard is built upon three key design modules: i) *rewriting boundary control* to limit sensitivity amplification, ii) *status-aware decision routing* to adapt rewriting under evolving privacy–utility conditions, and iii) *DP-aware Answering* to balance DP noise and result quality. Extensive experiments show that ReGuard consistently improves query answer quality under identical DP constraints and remains effective across a wide range of privacy budgets and query sensitivities. In particular, ReGuard reduces the mean relative error (MRE) by up to 74.6% compared to existing DP-enabled Text-to-SQL baselines.

1 Introduction

Driven by recent advances in large language models (LLMs), Text-to-SQL is rapidly emerging as a mainstream paradigm for database access through open-ended natural language interfaces. By automatically translating natural-language inputs into executable SQL, Text-to-SQL systems enable non-expert users to query databases without writing structured SQL queries. (Wang et al., 2025; Li et al., 2024). While this paradigm substantially lowers the barrier to data access, it fundamentally alters how query intent is specified, introducing ambiguity into query semantics. For example, natural-language queries may omit critical constraints, such as temporal scopes or precise entities, resulting in multiple plausible SQL interpretations. Consequently, to improve practical reliability, recent work has increasingly relied on query rewriting and reformulation for Text-to-SQL (Xiao et al., 2022; Huang et al., 2025; Mao et al., 2024), making query intent more explicit prior to SQL generation. While effective at improving executability and ambiguity resolution, query rewriting has become a core component of modern Text-to-SQL systems, fundamentally shaping how queries are interpreted and executed in practice.

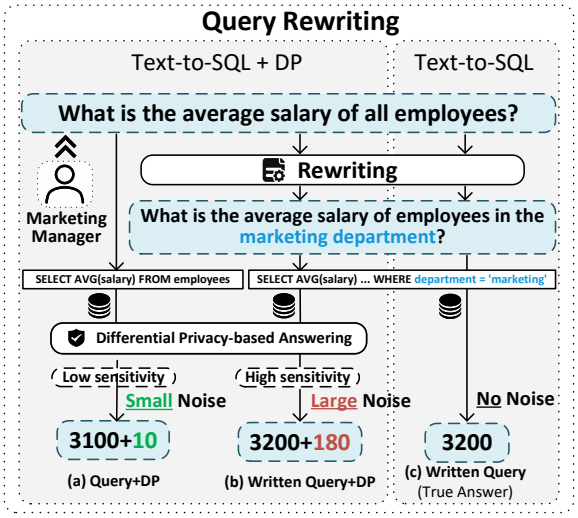

Figure 1: Query rewriting makes the query more specific (e.g., narrowing from all employees to the marketing department), but it also changes the DP query's effective cohort and thus its sensitivity. As illustrated, (b) may incur a higher sensitivity and require larger DP noise than (a), yielding a less accurate answer, even though the raw answer could be the same as (c).

As a core component that shapes query execution, query rewriting not only affects accuracy but also interacts with system-level requirements such

as privacy preservation. In particular, when Text-to-SQL systems are deployed over databases containing sensitive personal or proprietary records, protecting query answers becomes a practical requirement alongside query performance. A natural and widely adopted approach to enforcing such privacy constraints is differential privacy (DP) (McCartan et al., 2023; Google Cloud, 2023), which provides rigorous disclosure guarantees by adding noise to query answers. Under DP, the magnitude of added noise is calibrated to the query sensitivity, a key factor that determines how much information about individual records may be revealed (Dwork, 2006; El Ouadrhiri and Abdelhadi, 2022). In conventional SQL systems, DP mechanisms are typically applied to fixed query workloads, assuming that query semantics and sensitivity are determined prior to noise injection.

However, integrating DP into Text-to-SQL is non-trivial. Rewritten text often increases query clarity by introducing finer-grained predicates, narrower conditions, or more precise targets (Xiao et al., 2022; Mao et al., 2024; Chai et al., 2023), which directly amplifies the privacy risk of the query. As a consequence, DP mechanisms must react more aggressively (e.g., by injecting larger noise). In other words, gains from rewriting can be offset by stronger DP mechanisms, so rewriting may fail to improve and even reduce query performance in certain cases. As shown in Figure 1, query rewriting fundamentally influences not only query text but also the amount of noise required and the utility of the released answers; a phenomenon that is consistently observed in our empirical evaluation, as detailed in Section 4.2. Despite this influence, how query rewriting affects DP noise and utility has not been explicitly considered in existing Text-to-SQL pipelines, leaving utility optimization under DP constraints largely unexplored. Bridging this gap is crucial for safely and effectively enabling Text-to-SQL access to databases protected by differential privacy.

Motivated by these observations, we identify a previously unstudied problem at the intersection of semantic query rewriting and differential privacy. We formalize this problem setting as **Text-to-DP-SQL**. Specifically, in this setting, we consider the following problem: *given a natural-language query and a space of rewriting strategies, how can rewriting and DP-based query answering be jointly optimized to maximize utility under privacy constraints?* To address this problem, we propose **ReGuard**, a utility-aware Text-to-DP-SQL system built upon a key insight: the sensitivity increase induced by query rewriting is not monolithic, but arises from fundamentally different sources. Accordingly, ReGuard decomposes rewriting-induced sensitivity changes into two distinct categories. The first captures sensitivity shifts caused by semantic deviations introduced by rewriting strategies, while the second reflects sensitivity variations inherent to the underlying data distribution. This decomposition enables ReGuard to reason about rewriting decisions and DP noise in a fine-grained and principled manner. To control sensitivity amplification caused by semantic rewriting, ReGuard introduces a **Rewriting Boundary Control** module that explicitly constrains the extent of permissible rewriting, which regulates rewriting granularity to prevent excessive sensitivity growth. To account for sensitivity variations driven by data distribution, ReGuard incorporates a **Status-aware Decision Routing** module that conditions rewriting decisions on the current privacy–utility status. Finally, to ensure that rewriting decisions translate into actual utility improvements under DP, ReGuard employs a **DP-aware Answering** module that evaluates the decision gain and dynamically adopts the decisions. Our contributions are summarized as follows:

- We formalize a new problem setting, **Text-to-DP-SQL**, which integrates Text-to-SQL with differential privacy, and identify a fundamental challenge unique to this setting: query rewriting and DP-induced noise are inherently coupled due to open-ended natural-language inputs.

- We propose **ReGuard**, a utility-aware Text-to-DP-SQL system that resolves this challenge by explicitly disentangling rewriting-induced sensitivity effects and jointly optimizing rewriting decisions and DP-based query answering through three principled modules.

- We conduct extensive experiments on public benchmarks, demonstrating that ReGuard consistently improves answer utility under DP constraints. In particular, ReGuard achieves up to a 74.6% relative reduction in MRE and a 15.2% relative improvement in compliance compared to DP-enabled Text-to-SQL baselines that directly apply DP mechanisms.

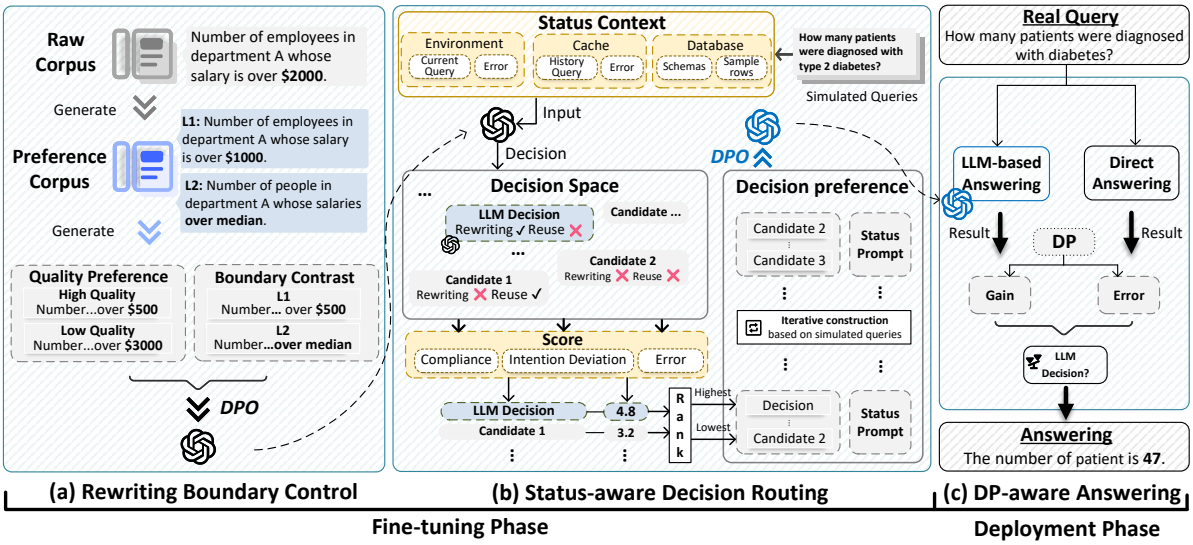

Figure 2: Architecture of ReGuard.

2 Preliminaries and Related Work

2.1 Preliminaries

Text-to-SQL. Let a Text-to-SQL model be $g : \mathcal{T} \times S \to \mathcal{Q}$, where $\mathcal{T}$ is the space of natural-language queries, $\mathcal{Q}$ is the space of executable SQL queries, and S denotes the database schema (optionally with external knowledge). Given a user query $t \in \mathcal{T}$, the model generates $q = g(t, S)$. Executing q on database D returns the answer $y = f_q(D)$.

Differential Privacy for SQL. We adopt record-level adjacency: two databases D and D' are adjacent (i.e., $D \sim D'$) if they differ in a single record. A randomized mechanism $\mathcal{M}$ satisfies (ε, δ)-DP if for any adjacent $D \sim D'$ and any output set O,

$$\Pr[\mathcal{M}(D) \in O] \leq e^{\varepsilon} \Pr[\mathcal{M}(D') \in O] + \delta. \quad (1)$$

In practice, a noisy answer released by $\mathcal{M}$ is associated with the query sensitivity Δ, which reflects the maximum change in the query output between two adjacent databases differing in one record, and thus serves as a key driver of privacy risk under DP. In this paper, we use the Gaussian mechanism (see Appendix A.1) to implement (ε, δ)-DP. Accordingly, we provide an explanation in Appendix A.2 of how queries influence Δ and the noise magnitude by the mechanism, and provide the privacy analysis of our system in Appendix A.3.

2.2 Differentially Private Query Answering

DP query answering systems release an approximate answer by injecting noise into the database result (Dwork et al., 2014). A line of database research has developed general-purpose DP query processing for relational workloads. For example, DOP-SQL (Yu et al., 2024) is a private SQL system designed to support a broad class of relational queries with high utility and extensibility. More recently, continual observation under DP has also been studied for relational operators such as joins, enabling repeated releases over time with improved accuracy guarantees (Dong et al., 2024). These DP query answering systems typically need to restrict users to a supported class of statistical queries (e.g., counts) (Dong et al., 2023; Stoisser et al., 2025).

2.3 Query Rewriting in Text-to-SQL

Query rewriting has been used for multiple objectives in Text-to-SQL, most prominently to improve task performance and robustness (e.g., disambiguation, completion, and schema/value grounding). For example, DART-SQL rewrites questions using database content to reduce ambiguity and further refines SQL with execution feedback (Mao et al., 2024), while MQA-SQL generates multiple rephrased variants and aggregates candidates based on consistency signals (Huang et al., 2025). Besides, rewriting and related mechanisms are also introduced to satisfy interface or application constraints. For example, STEPS (Tian et al., 2023) provides controllable intermediate explanations that users can validate and edit during SQL generation. LITHE (Dharwada et al., 2025) provides a rule-guided query rewriting with safeguards to enforce semantic correctness. More recently, SAFENLIDB (Liu et al., 2025b) aligns LLM-based NLIDB

with privacy-security preferences via synthesized security-reasoning interactions. Rewriting modifications can reshape what is being released, which in turn affects downstream DP mechanisms (e.g., sensitivity and the required noise). Existing works have not discussed this issue. We highlight and model this interaction influence, offering implications for deploying Text-to-SQL interfaces in privacy-preserving data systems.

3 ReGuard

Text-to-DP-SQL. In the differential privacy (DP)–based Text-to-SQL, the system ensures data privacy by releasing a perturbed query result $y = f_q(D)$:

$$\hat{y} = \mathcal{M}(y; \varepsilon, \delta)\,, \tag{2}$$

We call the DP-extended Text-to-SQL as **Text-to-DP-SQL**.

Problem Definition. We consider a deployment setting in which users only observe the final answers released by a DP mechanism. For each natural-language query t, a database owner specifies a constrained rewriting space $\mathcal{G}(t) \subseteq \mathcal{T}$, where $t \in \mathcal{G}(t)$, reflecting admissible rewrites under accuracy, safety, and disclosure requirements.

Given t, the system selects a rewritten query $t' \in \mathcal{G}(t)$, which can be expressed as

$$t' = \phi(t; \mathcal{G}, h), \tag{3}$$

where ϕ denotes a rewriting strategy parameterized by contextual information h. The rewritten query t' is then translated into an executable SQL query $q' = g(t', S)$ over schema S, producing a true query answer $y = f_{q'}(D)$ on database D. Finally, a DP mechanism is applied to y to generate the released answer observed by the user. Unlike conventional Text-to-SQL systems, a Text-to-DP-SQL system must account for how query rewriting affects query sensitivity under DP. Given a natural-language query t and its admissible rewriting space $\mathcal{G}(t)$, our goal is to design a principled answering system that generates a rewritten query $t' \in \mathcal{G}(t)$ to maximize overall utility under DP constraints. This objective can be formulated as:

$$\begin{aligned} &\max_{t' \in \mathcal{G}(t)} \; U(t', t, S, D) \\ &\quad \text{s.t. } \mathcal{M}(\cdot; t') \text{ is an } (\varepsilon, \delta)\text{-DP mechanism.} \end{aligned} \tag{4}$$

Here, $U(\cdot)$ denotes a composite utility function that captures multiple aspects of utility, including fidelity to the original query intent and robustness to DP-induced noise. The concrete instantiations of $U(\cdot)$ are introduced in Section 4.

Overview. Figure 2 illustrates the architecture of ReGuard, which leverages an LLM to optimize rewriting and DP release decisions via Direct Preference Optimization (DPO) jointly. We identify two primary sources of utility degradation under DP: (i) excessive semantic deviation introduced by aggressive rewriting, which increases query specificity and can amplify sensitivity and the required noise; and (ii) sensitivity fluctuations that arise even under mild semantic changes due to database distribution, making the noise scale unstable.

In the fine-tuning phase, **(a) Rewriting Boundary Control** trains the model to perform sensitivity-aware query rewriting. Specifically, we categorize rewrite operations into different sensitivity-impact levels (e.g., L_1/L_2), reflecting the extent to which a rewrite amplifies query sensitivity under DP. This categorization allows the model to distinguish between mild refinements and aggressive rewrites, and to internalize bounded rewriting behaviors. Optimized via DPO, the model is guided to prefer beneficial rewrites while avoiding over-aggressive refinements, thereby alleviating the source (i). **(b) Status-aware Decision Routing** trains the model to adaptively choose between rewriting a new query and reusing a compatible historical query based on the current runtime privacy-utility status. By substituting a new query with a suitable historical one, the system can avoid answering queries under unstable or excessively high DP noise. Optimized via a second-stage DPO, the model learns to leverage status information to route each query to the action with the highest expected utility under DP, thereby alleviating the source (ii).

In the deployment phase, **(c) DP-aware Answering** further chooses answering release mode between LLM-based and direct answering under DP by comparing utility gains and release error, which improves the robustness of the final release answer.

3.1 Rewriting Boundary Control

Even small textual refinements could affect query sensitivity and, consequently, the amount of DP noise required. We regulate the extent of text modifications by explicitly **stratifying** permissible rewriting strategies according to how they alter the executed query, and **select** the least-invasive strategy that satisfies the rewriting requirements.

Strategy Stratification. Permissible rewriting strategies are categorized into two atomic levels based on the scope of semantic change and the resulting sensitivity impact.

One is L_1**-level (Condition Rewriting)** that contains the rewriting strategies only adjusting *query predicates or filters*, such as bucketization, canonicalization, or range relaxation, while preserving the target statistic and overall query intent. The other is L_2**-level (Semantic Rewriting)** that contains the rewriting strategies modifying the *core query semantics*, such as enforcing aggregation or cohorting, substituting the target statistic, or restricting join paths—to meet disclosure constraints.

This stratification reflects that different rewriting strategies affect a query to different extents. L_1-level strategies primarily adjust cohort definition and selectivity, whereas L_2-level strategies could additionally change what is being measured (e.g., target statistic). Such changes can often induce larger and more variable sensitivity under DP.

Strategy Selection and Rewriting. We further leverage an LLM to analyze each query and perform the appropriate rewriting operation according to the predefined stratification through DPO. Specifically, we first generate a diverse natural-language **raw corpus** across multiple domains. For each query, we then produce two minimally compliant rewrite candidates, one following the L_1-level and the other following the L_2-level to generate **a preference corpus**. We then construct two types of pair for DPO. *Boundary-contrast* pairs encourage the model to distinguish and choose appropriate strategies, while *quality-preference* pairs encourage the model to generate high-quality rewrites within each level. Details of the data generation procedure and prompts are provided in Appendix B.1.

3.2 Status-aware Decision Routing

Beyond errors induced by rewriting itself, the data distribution can also introduce sensitivity instability: query sensitivity could fluctuate due to join paths or other SQL conditions. A straightforward remedy is to iteratively rewrite high-sensitivity queries until finding a suitable variant, which could incur unpredictable computation and latency.

Instead, we draw inspiration from DP-based data stream release mechanisms (Ren et al., 2022) and prior Text-to-SQL systems that leverage contextual information (Liu et al., 2025a; Zhang et al., 2024; Pourreza and Rafiei, 2023). We adopt a simple yet efficient strategy: when rewriting would significantly increase sensitivity, we require the LLM to determine whether a substitutable historical query with low sensitivity exists. While this reuse strategy offers the LLM an end-to-end query rewriting and sensitivity adaptation capability, it requires providing sufficient **decision context** and explicitly specifying the **decision criterion** (e.g., how to select the best candidate) to guide the LLM.

Decision Context. We provide the LLM with a structured **status information**. Specifically, we first supply database context, including field-level schemas and a small number of sampled rows per table, to constrain the feasible rewrite content. To support LLM substitution judgment, we then incorporate the current query along with its DP expected error as environment context. Rather than exposing raw continuous values, we discretize the expected noise into ordinal bands to improve judgment stability. Finally, we maintain a cache of recent query–expected error pairs (query, error) and retrieve the top-n most semantically similar historical records for the current query. These components are encoded in a status prompt that guides the model's decision.

Decision Criterion. Based on the above analysis, we define the LLM **decision space** including three actions: (i) *direct answering* by applying the DP mechanism to the current query, (ii) *rewriting* followed by DP answering, and (iii) *history reuse* by substituting the current query with a historical query. However, it is hard to describe the decision criterion of these three ways at a specific prompt, since it involves balancing multiple (e.g., DP-induced error) competing factors under complex status information. To operationalize this criterion, we continue to embed the criterion implicitly through DPO.

Specifically, we generate a series of simulated queries (see Appendix C.2 for the construction process) for the DPO preference-pair dataset. To ensure comparability among all actions, we ask the LLM with the status prompt to select one action from the space as the *anchor* decision for each query. Then, we generate the remaining candidate decisions for comparison: (i) if LLM decision does not select the *rewriting* action, we generate a *rewrite candidate* according to *rewriting* action; (ii) if LLM decision does not select the *history reuse* action, we generate a *rewrite candidate* by sampling a random entry from the history cache; (iii) if LLM

decision does not select the *direct answering* action, we create a *default candidate* according to *direct answering* action. Details on the construction of the status prompt, the generation of the *rewrite candidate* are provided in the appendix B.2.1.

Then, we compute a joint utility score for each candidate decision:

$$R = R_{com} + R_{sim} + R_{err}. \tag{5}$$

R_{com} reflects compliance of a rewritten query. R_{int} reflects the intention deviation, and is computed as follows,

$$R_{int} = \cos(\text{emb}(t), \text{emb}(t')), \tag{6}$$

where $\cos(\cdot)$ is the cosine similarity function. $\text{emb}(t)$ and $\text{emb}(t')$ denote the embedding of the original query t and the rewritten query t', respectively. R_{err} discretize the expected error into k levels and assign a score accordingly:

$$R_{\text{err}} = \mathbf{s}^\top \mathbf{1}_{\ell(e_{t'})}, \tag{7}$$

where $\mathbf{s} = \{s_1, s_2, ...s_k\}$ and $\ell(e_{t'}) \in \{1, \ldots, k\}$ maps the expected error $e_{t'}$ of t' to its discretized level score, and $\mathbf{1}_{\ell(e_{t'})} \in \{0, 1\}^k$ is the corresponding one-hot indicator. Finally, we select the highest-scoring and lowest-scoring candidates to form a **decision preference** pair. Details on the computation of R_{com}, R_{err} and expected error are provided in the appendix B.2.2.

3.3 DP-aware Answering

LLM decisions are driven by learned preferences rather than a precise sensitivity calculation, which could occasionally trigger decisions with no effectiveness. Therefore, we introduce a calibration step in the actual query answering to **compare and select** between the two answering release modes: (i) *Direct Answering*, release following the *direct answering* action, which incurs the **error cost** induced by noise; and (ii) *LLM-based Answering*, release following the LLM decision, which obtains the **gain** from *rewriting* or *history reuse*.

Cost of *Direct Answering*. The error cost of *Direct Answering* is measured by:

$$C = \frac{|y_t - \hat{y}_t|}{y_t}, \tag{8}$$

where y_t and $\hat{y}_t$ is the raw answer and perturbed answer of t from *Direct Answering*, respectively. Equation 8 captures the relative error between the raw answer and the perturbed answer.

Gain of *LLM-based Answering*. The gain is measured by:

$$G = \lambda\, B_{\text{saved}}(t) + \eta\, \frac{R(t') - R(t)}{R(t)}, \tag{9}$$

where $B_{\text{saved}}(t)$ is the privacy budget saved by history reuse (and is 0 if history is not reused), representing the contribution to the cost savings, which is meaningful for budget-limited scenarios. $R(\cdot)$ denotes the utility score calculated by Equation (5), $\frac{R(t')-R(t)}{R(t)}$ captures the relative improvement of the score introduced by the LLM decision, and λ, η are weighting coefficients. The selection follows rule:

$$\mathrm{D}(t) = \begin{cases} \textit{Direct Answer}, & \text{if } C \geq G, \\ \textit{LLM-based Answer}, & \text{otherwise.} \end{cases} \tag{10}$$

4 Experiments

4.1 Settings

Datasets. We evaluate our approach on three datasets. **Spider** (Yu et al., 2018) and **BIRD** (Li et al., 2023) are cross-domain Text-to-SQL benchmarks with multi-table schemas and complex SQL queries. **Adult** (Becker and Kohavi, 1996) is a classic tabular dataset with sensitive demographic and income attributes. For detailed construction procedures of rewriting workload for DP-based evaluation and usage settings of the dataset, please refer to Appendix C.1.

Evaluation Metrics. Execution accuracy (EX) is no longer a sufficient metric, as DP introduces noise. We therefore replace EX with two complementary DP-aware metrics: (i) *Compliance Score (CS)*, which measures whether the released query satisfies prescribed rewriting rules, and (ii) *Mean-relative error (MRE)*, which quantify the utility loss of DP-released results relative to the true answers. Apart from that, *Semantic Similarity (SS)* quantifies the semantic faithfulness of rewriting by cosine similarity. *Adoption Rate (AR)* reports the adoption rate of LLM interventions for methods with a posterior selection step. *Budget Usage (BU)* reports the fraction of the total privacy budget consumed, representing the utilization of budget.

Baseline Comparison. We adpot the following for comparison, including: **(i) Direct-DP** that applies the DP mechanism directly to the SQL result generated from the original query, reflecting the most common deployment in the Text-to-SQL

Method	Adult					Spider					BIRD				
	MRE↓	CS↑	SS↑	BU↑	AR∼	MRE↓	CS↑	SS↑	BU↑	AR∼	MRE↓	CS↑	SS↑	BU↑	AR∼
Direct-DP	7.96	0.46	-	-	-	28.91	0.48	-	-	-	10.92	0.49	-	-	-
Compliance-DP	7.68	0.41	0.92	-	-	15.60	0.45	0.90	-	-	11.61	0.45	0.90	-	-
SimHist-DP	9.73	0.48	0.90	-	0.70	34.52	0.50	0.91	-	0.77	14.43	0.51	0.90	-	0.73
Llama-8B	4.39	0.47	0.94	0.03	0.97	17.07	0.51	0.93	0.01	0.99	17.83	0.48	0.94	0.08	0.98
Mistral-7B	3.08	0.47	0.95	0.01	0.18	8.79	0.49	0.97	0.01	0.20	12.19	0.50	0.96	0.02	0.16
GPT-5.2	3.07	0.48	0.91	0.01	0.99	7.87	0.51	0.94	0.02	0.91	12.23	0.52	0.96	0.01	0.92
Gemini-3-Pro	4.28	0.49	0.93	0.01	0.99	7.66	0.52	0.96	0.01	0.97	12.06	0.52	0.94	0.02	0.99
ReGuard	**2.63**	**0.53**	**0.96**	**0.26**	0.40	**7.34**	**0.56**	**0.97**	**0.16**	0.30	**10.63**	**0.55**	**0.98**	**0.12**	0.27

Table 1: Overall performance of different methods on the three datasets. ↑ / ↓ indicate better direction; ∼ denotes descriptive metrics.

tasks; **(ii) Compliance-DP** that rewrites t into a compliant query under the prescribed L_1/L_2 restriction, and then apply the DP mechanism; **(iii) SimHist-DP** that retrieves the most similar candidate from the release history using cosine similarity(≥ 0.8) or it falls back to Compliance-DP; and **(iv) LLM-Policy** that prompts an LLM to output a decision conditioned on the status prompt, and then execute the DP release accordingly, which contains models: two open-source instruction-tuned LLMs, Meta-Llama-3.1-8B-Instruct (Llama-8B) and Mistral-7B-Instruct-v0.3 (Mistral-7B), and two commercial closed-source general-purpose LLMs, GPT-5.2 and Gemini-3-Pro.

In ReGuard, we use Qwen2.5-7B-Instruct as the backbone model and conduct two-stage DPO fine-tuning. Details of the metric implementation and hyperparameters are provided in Appendix C.3.

4.2 Overall Performance

Accuracy. Table 1 summarizes the overall performance of different methods across three datasets. Across all three datasets, our framework consistently achieves the lowest MRE. On Adult, ReGuard reduces MRE to 2.63, corresponding to a 67.0% relative reduction compared to Direct-DP (7.96) and it also improves upon strong LLM-based policies. Similar trends are observed on Spider and BIRD, where ReGuard consistently achieves the lowest MRE, with relative reductions of up to 75% on Spider compared to direct DP baselines.

Preference Compliance and Semantic Preservation. Beyond accuracy, ReGuard consistently achieves the highest CS across all datasets. On Adult, ReGuard improves CS to 0.53, corresponding to a 15.2% relative gain over Direct-DP and a 29% gain over Compliance-DP. Similar improvements are observed on Spider and BIRD. Meanwhile, semantic similarity (SS) remains consistently high, matching or exceeding strong LLM-based policies.

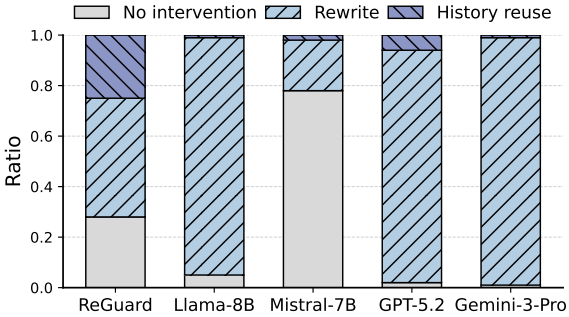

Figure 3: Decision distribution on the Spider dataset.

Why is ReGuard's Decision Policy Effective? To better understand the source of ReGuard's performance gains, we analyze the distribution of decision types produced by different models, as illustrated in Figure 3. The results reveal that decision behaviors are highly model-dependent. Most LLM-based policies exhibit a strong bias toward rewriting, with rewrite ratios exceeding 90% for Llama-7B, GPT-5.2, and Gemini-3-Pro. While aggressive rewriting can improve constraint satisfaction (CS) by enforcing query compliance, it overlooks the complementary benefits of other actions. In contrast, ReGuard maintains more balanced allocations among the three actions, with substantial proportions of no intervention and history reuse.

4.3 Robustness Evaluation

Robustness to Privacy Budget Variation. As shown in Figure 4, all methods exhibit a general trend of decreasing MRE as ε increases. Our method consistently achieves the lowest MRE under all budget settings, and demonstrates a smooth and monotonic improvement as ε increases (from 1.48 at $\varepsilon = 2$ to 0.46 at $\varepsilon = 8$).

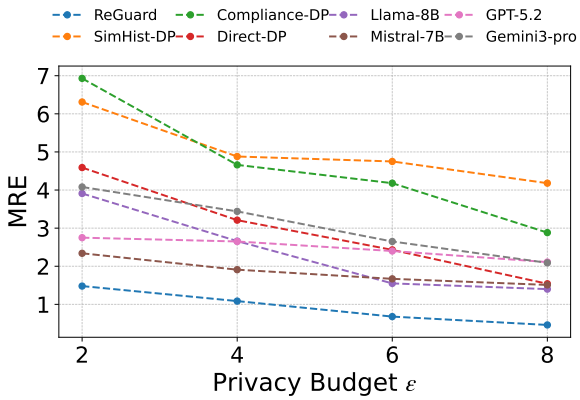

Figure 4: Result of varying ϵ on the Adult dataset.

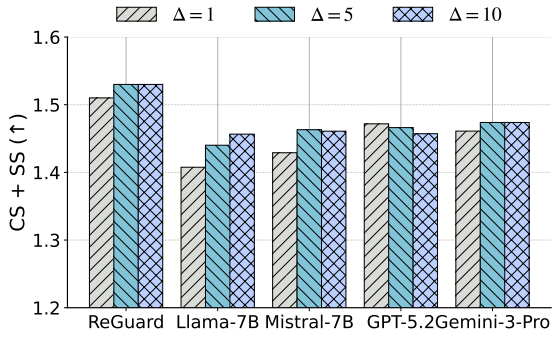

Figure 5: Result of varying Δ on the BIRD dataset.

Robustness to Sensitivity Variation. As shown in Figure 5, we observe that LLM-based policies tend to exhibit larger fluctuations across sensitivity settings. In contrast, ReGuard remains stable: CS+SS stays consistently high with only minor variation as sensitivity increases.

4.4 Ablation Studies

Ablation of Core Modules. Table 2 reports the impact of removing each core module on the Spider dataset. Removing SDR increases MRE, indicating its key role in controlling the error under DP. Removing results the BC in a moderate increase in MRE and noticeable drops in both CS and SS, indicating its the role to prevent violating preference constraints and introducing semantic drift. When removing the URE, the system achieves slightly higher budget usage but at the cost of increased MRE and reduced CS.

Ablation on Weight Parameters. As shown in Appendix C.4, we study the effect of varying the trade-off weights λ and η in the utility function. Fixing $\eta = \eta_0$ and increasing λ gradually shifts the system toward budget-oriented decisions. When fixing $\lambda = \lambda_0$ and varying η, we observe that moderate values of η improve accuracy while maintaining reasonable budget usage, whereas overly large η leads to marginal gains in MRE at the expense of higher adoption rates.

Method	MRE ↓	CS ↑	SS ↑	BU ↑	AR ∼
ReGuard	**5.91**	**0.56**	**0.97**	0.18	0.21
w/o RBC	6.08	0.54	0.96	0.23	0.25
w/o SDR	15.53	0.56	0.90	0.00	1.00
w/o DA	6.20	0.54	0.95	**0.27**	0.30

Table 2: Ablation study of modules on the spider dataset. RBC: Rewriting Boundary Control; SDR: Status-aware Decision Routing; DA: DP-aware Answering.

Ablation on Joint Score. As detailed in Appendix C.5, using a single component alone leads to imbalanced decision behavior: while R_{com} or R_{sim} improves CS and SS, it often over-encourages rewriting and results in suboptimal budget utilization, as reflected by extreme BU and AR values. In contrast, integrating all three components consistently yields a more balanced trade-off, achieving competitive MRE and stable CS/SS while avoiding over-aggressive interventions.

Why is ReGuard Robust? ReGuard explicitly balances semantic quality, noise-induced error, and budget efficiency. As shown in the ablation study on the joint score (Appendix C.5), relying on any single reward component leads to biased decision behavior. In contrast, integrating R_{com}, R_{sim}, and R_{err} enables ReGuard to capture complementary aspects of the decision objective. This balanced objective is also reflected in the resulting decision distribution. Further evidence comes from the ablation of core system modules (Table 2), where the complementary roles of individual components are critical to robustness.

5 Conclusion

We show that query rewriting and differential privacy are inherently coupled in Text-to-SQL systems, and that unregulated rewriting can harm utility under DP. To formalize this setting, we introduce *Text-to-DP-SQL*, which integrates open-ended natural language querying with DP-protected answering release. ReGuard addresses this issue by jointly controlling rewriting and answering decisions in a utility-aware manner, yielding consistent improvements across datasets.

6 Limitations

Despite the effectiveness of ReGuard, our work has several limitations:

First, our system focuses on a restricted class of DP-compatible SQL workloads, primarily aggregation and counting-style queries. Although this setting aligns with common DP deployments and allows controlled evaluation, extending ReGuard to more expressive query classes (e.g., nested queries or non-linear analytics) would require additional sensitivity modeling and is left for future work.

Second, the decision policy is learned under simulated workloads and system configurations. While our experiments demonstrate strong generalization across datasets and models, real-world deployments may involve different data distributions, schema characteristics, or privacy policies. Adapting the decision model online or jointly training it with deployment-specific feedback remains an open problem.

Finally, ReGuard introduces additional system complexity and computational overhead, including LLM-based decision making and post-hoc utility evaluation. Although these costs are modest compared to SQL execution and DP mechanisms in our experiments, latency-sensitive applications may require further optimization or lightweight approximations.

7 Ethical Considerations

This work aims to improve the safe and responsible deployment of natural-language database interfaces under differential privacy guarantees. By explicitly modeling how query rewriting interacts with privacy-induced noise, ReGuard seeks to prevent unintended disclosure risks and misleading outputs that could arise from uncalibrated rewriting under DP constraints.

Our system does not weaken differential privacy guarantees. All released answers are produced by standard DP mechanisms, and rewriting or history reuse decisions rely on the post-processing immunity property of DP, which ensures that privacy loss is not increased beyond the original release. However, we acknowledge that misuse or misconfiguration of rewriting rules could lead to semantic misinterpretation of user intent, potentially affecting fairness or decision-making downstream. To mitigate this risk, ReGuard emphasizes controllable rewriting boundaries and utility-based validation, but responsible deployment still requires careful policy design and human oversight. We also note that large language models used in our system may reflect biases present in their training data, which could influence rewriting behavior or decision preferences.

We encourage future work to further explore transparency, accountability, and user control in privacy-preserving natural-language data access systems.

References

Barry Becker and Ronny Kohavi. 1996. Adult. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5XW20.

Linzheng Chai, Dongling Xiao, Zhao Yan, Jian Yang, Liqun Yang, Qian-Wen Zhang, Yunbo Cao, and Zhoujun Li. 2023. Qurg: Question rewriting guided context-dependent text-to-sql semantic parsing. In *Pacific Rim International Conference on Artificial Intelligence*, pages 275–286. Springer.

Sriram Dharwada, Himanshu Devrani, Jayant Haritsa, and Harish Doraiswamy. 2025. Query rewriting via llms. *Preprint*, arXiv:2502.12918.

Wei Dong, Zijun Chen, Qiyao Luo, Elaine Shi, and Ke Yi. 2024. Continual observation of joins under differential privacy. *Proc. ACM Manag. Data*, 2(3).

Wei Dong, Dajun Sun, and Ke Yi. 2023. Better than composition: How to answer multiple relational queries under differential privacy. *Proc. ACM Manag. Data*, 1(2).

Cynthia Dwork. 2006. Differential privacy. In *International colloquium on automata, languages, and programming*, pages 1–12. Springer.

Cynthia Dwork, Aaron Roth, and 1 others. 2014. The algorithmic foundations of differential privacy. *Foundations and trends® in theoretical computer science*, 9(3–4):211–407.

Ahmed El Ouadrhiri and Ahmed Abdelhadi. 2022. Differential privacy for deep and federated learning: A survey. *IEEE access*, 10:22359–22380.

Google Cloud. 2023. Differential privacy in bigquery. `https://cloud.google.com/bigquery/docs/reference/standard-sql/differential-privacy`. Accessed: 2025-01.

Yiming Huang, Jiyu Guo, Jichuan Zeng, Cuiyun Gao, Peiyi Han, and Chuanyi Liu. 2025. Mqa-sql: Mitigating question ambiguity in text-to-sql with multi-model collaboration and multi-variant query rephrasing. In *CCF International Conference on Natural Language Processing and Chinese Computing*, pages 289–300. Springer.

Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024. Codes: Towards building open-source language models for text-to-sql. *Proc. ACM Manag. Data*, 2(3).

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Ma Chenhao, Guoliang Li, Kevin Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. In *Advances in Neural Information Processing Systems*, volume 36, pages 42330–42357. Curran Associates, Inc.

Geling Liu, Yunzhi Tan, Ruichao Zhong, Yuanzhen Xie, Lingchen Zhao, Qian Wang, Bo Hu, and Zang Li. 2025a. Solid-SQL: Enhanced schema-linking based in-context learning for robust text-to-SQL. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 9793–9803, Abu Dhabi, UAE. Association for Computational Linguistics.

Ruiheng Liu, XiaoBing Chen, Jinyu Zhang, Qiongwen Zhang, Yu Zhang, and Bailong Yang. 2025b. Safenlidb: A privacy-preserving safety alignment framework for llm-based natural language database interfaces. *arXiv preprint arXiv:2511.06778*.

Wenxin Mao, Ruiqi Wang, Jiyu Guo, Jichuan Zeng, Cuiyun Gao, Peiyi Han, and Chuanyi Liu. 2024. Enhancing text-to-sql parsing through question rewriting and execution-guided refinement. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 2009–2024.

Cory McCartan, Tyler Simko, and Kosuke Imai. 2023. Making Differential Privacy Work for Census Data Users. *Harvard Data Science Review*, 5(4). Https://hdsr.mitpress.mit.edu/pub/y6oilhhe.

Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. In *Advances in Neural Information Processing Systems*, volume 36, pages 36339–36348. Curran Associates, Inc.

Xuebin Ren, Liang Shi, Weiren Yu, Shusen Yang, Cong Zhao, and Zongben Xu. 2022. Ldp-ids: Local differential privacy for infinite data streams. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1064–1077, New York, NY, USA. Association for Computing Machinery.

Josefa Lia Stoisser, Marc Boubnovski Martell, Kaspar Märtens, Lawrence Phillips, Stephen Michael Town, Rory Donovan-Maiye, and Julien Fauqueur. 2025. Query, don't train: Privacy-preserving tabular prediction from ehr data via sql queries. *Preprint*, arXiv:2505.21801.

Yuan Tian, Zheng Zhang, Zheng Ning, Toby Jia-Jun Li, Jonathan K. Kummerfeld, and Tianyi Zhang. 2023. Interactive text-to-SQL generation via editable step-by-step explanations. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 16149–16166, Singapore. Association for Computational Linguistics.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, LinZheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025. MAC-SQL: A multi-agent collaborative framework for text-to-SQL. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 540–557, Abu Dhabi, UAE. Association for Computational Linguistics.

Dongling Xiao, LinZheng Chai, Qian-Wen Zhang, Zhao Yan, Zhoujun Li, and Yunbo Cao. 2022. CQR-SQL: Conversational question reformulation enhanced context-dependent text-to-SQL parsers. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 2055–2068, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Jianzhe Yu, Wei Dong, Juanru Fang, Dajun Sun, and Ke Yi. 2024. Dop-sql: A general-purpose, high-utility, and extensible private sql system. *Proceedings of the VLDB Endowment*, 17(12):4385–4388.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task.

Hanchong Zhang, Ruisheng Cao, Hongshen Xu, Lu Chen, and Kai Yu. 2024. Coe-sql: In-context learning for multi-turn text-to-sql with chain-of-editions. *Preprint*, arXiv:2405.02712.

A Differential Privacy

A.1 Gaussian Mechanism

Differential privacy provides a formal guarantee that the inclusion or exclusion of a single individual in a dataset has a limited impact on the released output. Among various DP mechanisms, the Gaussian mechanism is widely used for real-valued queries and serves as the foundation of our privacy-preserving release.

Definition A.1 (Gaussian Mechanism (Dwork et al., 2014)) Let $f : \mathcal{D} \to \mathbb{R}^k$ be a query function with ℓ_2-sensitivity

$$\Delta_2(f) = \max_{D \sim D'} \|f(D) - f(D')\|_2,$$

where $D \sim D'$ denotes two adjacent datasets differing in one record. Given any function f: $\mathbb{N}^{|\mathcal{X}|} \to \mathbb{R}^k$, the Gaussian mechanism releases

$$M_G(x, f(\cdot), \epsilon, \delta) = f(x) + \mathcal{N}(0, \sigma^2),$$

where independent Gaussian noise with standard deviation σ is added to each dimension.

For a single invocation, the Gaussian mechanism satisfies (ε, δ)-differential privacy when the noise standard deviation σ is chosen as

$$\sigma \geq \Delta_2(f) \frac{\sqrt{2\ln(1.25/\delta)}}{\varepsilon}.$$

A.2 Implications for Text-to-SQL.

In Text-to-SQL systems, rewriting operations may alter query structure, predicate granularity, or join multiplicity, thereby affecting the effective sensitivity $\Delta_2(f)$. Under a fixed privacy budget (ε, δ), the injected noise standard deviation scales linearly with the sensitivity, i.e., $\sigma \propto \Delta_2(f)$. As a result, rewriting-induced sensitivity changes directly translate into different noise magnitudes and, consequently, different levels of noise-induced error in the released results. This observation motivates our system, which explicitly accounts for rewriting influences to the DP noise.

A.3 Privacy Analysis.

Lemma 1 (Serial Composition of Privacy (Dwork et al., 2014)). *Let $\mathcal{M}_i : \mathbb{N}^{|\mathcal{X}|} \to \mathcal{R}_i$ be (ϵ_i, δ_i)-differentially private for each $i \in [k]$. Define the composed mechanism $\mathcal{M}^{[k]} : \mathbb{N}^{|\mathcal{X}|} \to \prod_{i=1}^{k} \mathcal{R}_i$ by*

$$\mathcal{M}^{[k]}(x) = \big(\mathcal{M}_1(x), \ldots, \mathcal{M}_k(x)\big).$$

Then $\mathcal{M}^{[k]}$ is $\left(\sum_{i=1}^{k} \epsilon_i,\ \sum_{i=1}^{k} \delta_i\right)$-differentially private.

Theorem 1 (Privacy Guarantee of ReGuard). *For any sequence of queries processed by ReGuard, where i-th query are allocated (ϵ_i, δ_i), then the system satisfies $(\sum \epsilon_i, \sum \delta_i)$-differential privacy with respect to the underlying database.*

Proof. ReGuard outputs either (a) a newly released answer produced by invoking a DP mechanism, or (b) a previously released DP-protected answer (history reuse).

New answers. For each non-reuse step i, ReGuard invokes $\mathcal{M}_i$, which is (ϵ_i, δ_i)-DP by assumption.

History reuse. If ReGuard reuses a previously released answer, it does not issue any new query to the database and outputs an existing DP-protected value. Thus, reuse incurs no additional privacy loss and does not consume extra budget.

Decision process. The LLM-based decision rule uses internal signals (e.g., an expected error) to choose between reuse and invoking $\mathcal{M}_i$. The decision process does not issue any additional queries to the database and, these internal signals are never released to the user.

Therefore, the observable result of ReGuard consists of a sequence of DP release answers from mechanisms $\{\mathcal{M}_i\}$ interleaved with reuse steps. According to the Lemma 1, the overall privacy loss is $(\sum_i \epsilon_i, \sum_i \delta_i)$. If the accountant ensures $\sum_i \epsilon_i \leq \epsilon$ and $\sum_i \delta_i \leq \delta$, the system satisfies (ϵ, δ)-DP. □

B Implementation Details

B.1 Rewriting Boundary Control

To explicitly control rewriting behavior across and within rewriting levels, we construct two types of preference supervision: boundary-contrast pairs and quality-preference pairs.

Boundary-contrast pairs. Boundary-contrast pairs aim to teach the model *which rewriting level* should be applied under different rewriting requirements. For each original query t, we use GPT-5 to select a suitable target rewriting category c from the target workload, i.e., the category whose trigger and transformation description best matches the rewriting requirement (see Appendix D.1), and then generate the corresponding rewrite $t_{k,c}$, where k denotes the rewriting level of category c. We construct a contrastive pair by sampling another category c' from a *different* rewriting level $k' \neq k$, and generating the rewrite $t_{k',c'}$ using the same generator (GPT-5). Using level-specific prompts, we form a preference instance (x, y^+, y^-), where x includes the original query t, and $y^+ = t_{k,c}$, $y^- = t_{k',c'}$. This cross-level contrast provides an explicit training signal for boundary awareness, encouraging the model to choose the *least-invasive* transformation that satisfies the intended rewriting type, and to avoid both under-rewriting and over-rewriting. The example data are provided in appendix E.1.

Quality-preference pairs. Quality-preference pairs focus on *intra-level* discrimination. We sample multiple candidate rewrites for the same query using the corresponding level-specific prompt and an LLM generator (Qwen3). We then score these candidates with GPT-5 as an *LLM-as-a-Judge* (see appendix D.2), ranking them by overall rewrite quality (rule compliance, semantic faithfulness, and transformation minimality). The top-ranked

candidate is treated as the preferred rewrite and the bottom-ranked one as the dispreferred rewrite, forming a quality-preference pair. This intra-level preference supervision encourages the model to produce the most faithful rewrite among valid alternatives within the same boundary. The example data are provided in appendix E.2.

B.2 Online State-aware Decision

B.2.1 Action Representation.

Model decision space. The model selects an action from three high-level choices, *rewrite*, *reuse_history* and *None*. The action determines which argument is required: for *rewrite*, the model outputs the rewritten text; for *reuse_history*, it outputs the index of a cached query to reuse. We represent the decision using the following JSON schema:

```
{
 "action": "rewrite | reuse_history
 | None ",
 "rewrite_text": "...",
 // only if action == "rewrite"
 "history_index": 3
 // only if action == "reuse_history"
}
```

The status prompt used to guide the model's decision is shown in Appendix D.4.

Generating *rewrite* candidates. We use the rewriting model trained by the *Rewriting Boundary Control* module and rewrite generation prompt (see Appendix D.3) to generate the *rewrite* candidate. Concretely, we include the *Database information* component from the status prompt in Appendix D.4, and then generate the rewritten text. For each candidate, the corresponding decision space also changes accordingly.

B.2.2 Action Evaluation.

Computing R_{com}. We compute R_{com} using a reward model fine-tuned from the preference data generated by Rewriting Boundary Control. Specifically, the boundary-contrast and quality-preference pairs are used to construct a preference dataset, on which we fine-tune a lightweight reward model based on Qwen2.5-3B. This reward model serves as a proxy for the data owner's rewriting preference, capturing which rewriting behaviors are considered acceptable and beneficial under deployment constraints, and assigning higher scores to rewrites that better align with these preferences.

Computing R_{err}. The expected error e_t induced by releasing a query t is approximated using a proxy function $E(t)$, meaning that $e = E(t)$. Specifically, $E(\cdot)$ denotes the effective noise scale, i.e., the noise standard deviation σ induced by query t under the Gaussian mechanism (see Appendix A.1). Although the variance of a Gaussian distribution is unbounded in theory, we categorize the noise scale into different regimes for analysis in practice. Specifically, we truncate the variance to the range $[1, 10]$ and uniformly partition it into k disjoint intervals, each corresponding to a score.

C Details of Experiment

C.1 Workload

We construct our evaluation workload by instantiating the rewriting primitives summarized in Table 3, which captures commonly used rewriting strategies in prior SQL systems and is adopted as the basis for our evaluation. Concretely, we first collect linear counting queries already supported by the datasets, and treat them as seed templates. For the Spider dataset, we reserve 30% of the templates for constructing simulated queries in the subsequent section, and use the remaining templates for workload instantiation. We then generate additional query instances by systematically applying our rewriting operations. For example, for a canonicalized predicate pattern such as "older than 30" (mapped to our L1-level predicate rewriting), we produce multiple instantiated variants by varying constants and ranges (e.g., "older than 40", "older than 50") while keeping the underlying intent and schema consistent. We apply similar instantiation to other eligible predicate patterns and counting-style formulations, yielding a large set of linear queries with controlled semantic variations.

To further diversify the workload beyond template-based instantiation, we additionally leverage GPT-5 to synthesize extra natural-language queries that are compatible with each dataset schema and can be translated into the supported SQL forms for DP. We use the same taxonomy to filter and normalize these model-generated queries, ensuring that they are suitable for downstream DP evaluation. Overall, we curate a workload of 2,000 queries for each dataset to achieve broader coverage and diversity. For evaluation, we randomly sample 30 queries from the curated workload in each run and repeat this process 20 times. All reported results are averaged over these runs to

Rewriting Strategy	Level	Typical Trigger / Goal	Transformation (NL/SQL)
A. Accuracy-driven rewriting strategies			
Canonicalization	L1	Improve parseability/executability	Normalize operators/structures: "older than 30" → `age > 30`; "top 5" → `ORDER BY ... DESC LIMIT 5`
Disambiguation / Completion	L1	Resolve underspecification (scope, missing clauses)	Make intent explicit: "employees in sales" → "employees in the Sales department"; add missing `GROUP BY` / time scope
Schema / Value grounding	L3	Align NL mentions to schema/values (Spider/BIRD)	"NYC" → `city='New York City'`; "CS" → `dept='Computer Science'`
B. Safety-driven rewriting strategies			
Predicate bucketization	L1	Granularity adjustment (often for robustness/safety)	Replace exact predicates with bins/intervals: `age` $= 47 \rightarrow$ `age` $\in [45, 50]$; `date` $=' 2023\text{-}03\text{-}21' \rightarrow$ `date` $\in$ [Q1]
Range widening	L1	Overly narrow ranges / brittle filters	Relax tight ranges: `age` $\in [46, 47] \rightarrow$ `age` $\in [45, 50]$
Entity-to-cohort aggregation	L3	Entity-target queries; need cohort-level semantics	`metric`$(e) \rightarrow$ `AVG(metric) WHERE` $G = G(e)$ (e.g., `salary(e)`$\rightarrow$ `AVG(salary) WHERE dept=dept(e)`)
Target generalization	L3	Over-specific outputs; prefer stable statistics	`income`$(e) \rightarrow$ `MEDIAN(income)`; `amount`$(t) \rightarrow$ `HIST(amount)`; `is_refund`$(o) \rightarrow$ `refund_rate`

Table 3: **A** summarizes accuracy-driven rewriting strategies that are commonly studied in the Text-to-SQL. **B** summarizes safety-driven rewriting strategies that are more frequently explored in traditional database settings (e.g., company safety policy enforcement). We include **B** as a complementary taxonomy to broaden coverage and to verify that our system generalizes.

reduce variance and improve statistical reliability.

C.2 Simulated Query

Following the workload construction described in Appendix C.1, we use 30% of the Spider dataset to build the simulation workload, from which we generate a total of 800 simulated queries. For preference construction, we randomly sample 30 queries per round and repeat this process for 200 rounds, resulting in a preference-pair dataset of size 6,000.

C.3 Metric implementation and Hyperparameters

Metric implementation. CS is computed using a reward model fine-tuned on preference data derived from R_{com}, where higher scores indicate stronger compliance with prescribed rewriting rules. MRE is computed as the mean relative error between DP-released answers and their corresponding true answers. SS is measured as the cosine similarity between sentence embeddings of the original query and the rewritten query.

Hyperparameters. We set the total privacy parameter $\delta = 10^{-5}$ in each run. Unless otherwise specified, the privacy budget is fixed to $\varepsilon = 10$ for each query; in Section 4.2, we instead use $\varepsilon = 1$ to study performance under a stricter privacy setting. we also set $\lambda = 1$, $\eta = 10^3$, $k = 3$ and $n = 10$ in all experiments.

C.4 Ablation Study: Weight Parameters

(λ, η)	MRE↓	CS↑	SS↑	BU↑	AR∼
$(0, \eta_0)$	**4.33**	**0.53**	0.98	0.10	0.13
$(10\lambda_0, \eta_0)$	4.44	0.52	**0.99**	0.14	0.17
$(100\lambda_0, \eta_0)$	4.81	**0.53**	**0.99**	**0.19**	0.22
$(\lambda_0, 0)$	5.17	0.52	**0.99**	0.02	0.02
$(\lambda_0, 0.5\eta_0)$	4.19	**0.53**	0.98	0.10	0.14
$(\lambda_0, 10\eta_0)$	**3.96**	**0.53**	0.98	**0.12**	0.17

Table 4: Ablation on trade-off weights λ and η on the BIRD dataset.

C.5 Ablation Study: Joint Score

Table 5: Ablation on the joint score components in $R = R_{com} + R_{sim} + R_{err}$.

R_{pref}	R_{sim}	R_{err}	MRE↓	CS↑	SS↑	BU↑	AR∼
✓			3.71	0.57	0.95	0.38	0.44
	✓		3.97	0.55	0.99	0.09	0.11
		✓	3.23	0.55	0.96	0.21	0.26
✓		✓	5.71	0.56	0.94	0.31	0.34
✓	✓	✓	4.18	0.55	0.97	0.18	0.21

D Prompt Design

D.1 Category Selection Prompt

To construct boundary-contrast supervision, we first prompt GPT-5 to select a suitable rewriting category c from the target workload. The selection is performed by matching the rewriting requirement to each category's *trigger* and *transformation* description in our taxonomy. The candidate category list is retrieved from the Table 3 and `[QUERY]` is the user's query text.

Category Selection Prompt

You are given (i) an original query, and (ii) a list of candidate rewriting categories (from the target workload). Select the most suitable `category_id` **from the provided list**. A category is suitable if t can be rewritten to satisfy this category with the *least-invasive* transformation. *Least-invasive* means:
(1) preserve the user's intent and key constraints;
(2) do not remove essential filters unless required;
(3) avoid unnecessary semantic drift; and
(4) if multiple categories are feasible, choose the one requiring the fewest and smallest edits.
Use each category's **Typical Trigger/Goal** and **Transformation** description to make your decision.
Original query: [QUERY]
Candidate categories:
1) [category_id=..., level=..., trigger=..., transformation=...]
2) ...
Output: Return a `category_id`.

D.2 Rewrite Quality Evaluation Prompt

Rewrite Quality Evaluation Prompt

You are an expert evaluator for query rewriting in Text-to-SQL systems. Your task is to assess and rank multiple rewritten candidates of the same query produced under the *same rewriting category and level*.
All candidates are assumed to be *valid* with respect to the category constraints. Your goal is to identify which rewrite is of *higher overall quality*.
Evaluation Criteria:
Evaluate each candidate rewrite according to the following aspects:
1. **Rule Compliance**: Does the rewrite strictly follow the specified rewriting category and level constraints?
2. **Semantic Faithfulness**: Does the rewrite preserve the original user intent and essential constraints without introducing semantic drift?
3. **Transformation Minimality**: Among valid rewrites, does the rewrite apply the least-invasive modification, avoiding unnecessary edits or over-specification?
Important Notes: - All candidates belong to the same rewriting level (L1 or L2); do *not* compare across levels. - Do *not* favor rewrites that introduce additional constraints unless strictly required. - Minor surface differences are acceptable, but unnecessary semantic changes should be penalized.
Rewriting Context:
[category_id=..., level=..., trigger=..., transformation=...]
Original Query:
[QUERY]
Rewrite Candidates:
1) [REWRITE_1]
...
Output:
Rank the candidates from best to worst according to the evaluation criteria. Return only a ranked list of candidate identifiers (e.g., A > C > B).

D.3 Rewrite Generation Prompt

After selecting the target category c, we use GPT-5 with the rewrite-generation prompt to produce $t_{k,c}$;

for boundary contrast, we additionally generate $t_{k',c'}$ for a category c' from a different level $k' \neq k$.

Rewrite Generation Prompt

You need to rewrite queries according to a specified rewriting category and level constraints. Rewrite the following query according to the given rewriting category. You must strictly follow the level constraints:
Category Requirements:
[category_id=..., level=..., trigger=..., transformation=...]
Constraints:
- Apply the *least-invasive* change that satisfies the category. *Least-invasive* means:
(1) preserve the user's intent and key constraints;
(2) do not remove essential filters unless required;
(3) avoid unnecessary semantic drift; and
(4) if multiple categories are feasible, choose the one requiring the fewest and smallest edits.
- If level is L1: do *not* change the target statistic; only apply surface-level edits (e.g., predicate/value adjustments) allowed by the category.
- If level is L2: Target changes are allowed only if required by the category.
Input query: [QUERY]
Output: Return only the rewritten query.

D.4 Status Prompt

To support online state-aware decision making, we design a decision prompt which explicitly reasons over the current query, database information, and previously executed queries, and produces a structured decision in the form of a JSON output. This prompt is used both to generate decision examples for DPO training and during online inference. `[SCHEMA AND EXAMPLES]` includes table schemas (table and column names), primary and foreign key relations, and a small set of representative value examples for selected columns. `[CURRENT NOISE LEVEL]` presents noise level of current query derived from the noise multiplier. `[LIST OF EXECUTED QUERIES]` consists of previously executed natural-language queries, each annotated with its corresponding noise level derived from the noise multiplier.

Decision Prompt

Task. You are given a database query rewriting task with access to historical queries. Decide (i) whether the current query should be rewritten, and (ii) whether its result can be directly reused from history. Input Information:
Database information: [SCHEMA AND EXAMPLES]
Current query: [QUERY]
Current noise level: [CURRENT NOISE LEVEL]
Historical queries: [LIST OF EXECUTED QUERIES]
Step 1: Rewrite Decision. Determine whether the current query needs rewriting under the given level-specific constraints. If rewriting is required, provide a rewritten *natural language* statement that satisfies the constraints.
Step 2: Historical Substitutability Assessment. Based on the final statement from Step 1 (original or rewritten), rigorously judge whether its SQL result can be *directly replaced* by exactly one historical query result. *Direct replacement* means reusing a single historical result without any additional computation, deduction, or reliance on database information. If replacement is possible, output the corresponding historical query number.
Output: Return *only* a JSON object with fields: {`rewrite`, `rewritten_text`, `use_history`, `number`}.

E Example for Training Data

E.1 Quality-preference Pair Example

Quality-preference Pair Example

Prompt(x): Given (i) an original query t,and (ii) a list of candidate rewriting categories, select the *single* most suitable category and generate corresponding rewriting text.
Original query t: How many patients over 80 years old were admitted to ICU in 2022?
Candidate categories:
1) [category_id=..., level=..., trigger=..., transformation=...]
2) ...
Output: Return only a JSON object with fields: {`category_id`, `rewriting_text`}.

Chosen (y^+):

```
{
"category_id": "L1-Canonicalization",
"rewriting_text": "How many
   patients aged 80 or above were
   admitted to ICU in 2022?"
}
```

Rejected (y^-, drops an essential predicate):

```
{
"category_id": "L1-Canonicalization",
"rewriting_text": "How many patients
  were admitted to ICU in 2022?"
}
```

E.2 Boundary-contrast Pair Example

Boundary-contrast Pair Example

Prompt(x): Given (i) an original query t,and (ii) a list of candidate rewriting categories, select the *single* most suitable category and generate corresponding rewriting text.
Original query t: How many patients over 80 years old were admitted to ICU in 2022?
Candidate categories:
1) [category_id=..., level=..., trigger=..., transformation=...]
2) ...
Output: Return only a JSON object with fields: {`category_id`, `rewriting_text`}.

Chosen (y^+):

```
{
"category_id": "L1-Canonicalization",
"rewriting_text": "How many patients
aged 80 or above were admitted to ICU
in 2022?"
}
```

Rejected (y^-):

```
{
 "category_id": "L2-Target-
 Generalization",
"rewriting_text": "What is the age
distribution of patients admitted to
 ICU in 2022?"
}
```