MaskSQL: Safeguarding Privacy for LLM-Based Text-to-SQL via Abstraction

Anonymous Author(s)
Affiliation
Address
email

Abstract

Large language models (LLMs) have shown promising performance on tasks that require reasoning, such as text-to-SQL translation, code generation, and debugging. However, regulatory frameworks with strict privacy requirements constrain their integration into sensitive systems. State-of-the-art LLMs are also proprietary, costly, and resource-intensive, making local deployment impractical. Consequently, utilizing such LLMs often requires sharing data with third-party providers, raising privacy concerns and risking noncompliance with regulations. Although fine-tuned small language models (SLMs) can outperform LLMs on certain tasks and be deployed locally to mitigate privacy concerns, they underperform on more complex tasks such as text-to-SQL translation. In this work, we introduce MaskSQL, a text-to-SQL framework that utilizes abstraction as a privacy protection mechanism to mask sensitive information in LLM prompts. Unlike redaction, which removes content entirely, or generalization, which broadens tokens, abstraction retains essential information while discarding unnecessary details, striking an effective privacy–utility balance for the text-to-SQL task. Moreover, by providing mechanisms to control the privacy-utility tradeoff, MaskSQL facilitates adoption across a broader range of use cases. Our experimental results show that MaskSQL outperforms leading SLM-based text-to-SQL models and achieves performance approaching state-of-the-art LLM-based models, while preserving privacy.

1 Introduction

Structured databases are central to applications across science, business, healthcare, and government. However, retrieving information from these systems typically requires knowledge of Structured Query Language (SQL), creating a steep barrier for non-technical users. The text-to-SQL task bridges this gap by translating natural language (NL) questions into executable SQL queries, allowing users to interact with databases intuitively and without specialized expertise.

Recent advances in language models (LMs) have significantly improved the accuracy and availability of text-to-SQL solutions, enabling deployment across diverse domains and schemas. For example, the most performant approaches on the text-to-SQL benchmarks Spider [54], BIRD[26], and Spider 2.0 [22] rely on LMs as their backbone for SQL generation. An LM's performance typically scales with parameter count [9], leading to a common distinction [12] between small LMs (SLMs, < 10B parameters), and large LMs (LLMs, 10s to 100s of billions of parameters) which require specialized infrastructure to run. While LLMs dominate text-to-SQL benchmarks, due to their increased hardware requirements they are typically accessed via remote APIs hosted by specialized inference providers. These hosted APIs are attractive to users as they largely eliminate infrastructure outlays, but also introduce severe privacy concerns; passing data through third-party APIs exposes sensitive user and

Submitted to Workshop on Regulatable ML at the 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

schema information to heightened privacy risks [35, 29, 18]. For instance, recent work demonstrated that databases used in a text-to-SQL system are vulnerable to schema inference, where an adversary can reconstruct proprietary schema details by probing the system with carefully crafted queries [21]. These criticisms could lead to stricter standards for text-to-SQL tasks in privacy laws such as GDPR in Europe, HIPAA in the USA, or PIPEDA in Canada.

To illustrate the dilemma, consider the text-to-SQL task shown in Example 1. The user has two options to generate this query: (i) send their data to a powerful but *untrusted* LLM hosted remotely, with the risk of exposing schema and personally identifiable information (PII); or (ii) rely on *trusted* SLMs hosted locally, which often fail to handle complex SQL constructs such as nested queries, window functions, or common table expressions (see Figure 3 in Appendix E). Notable downsides exist with either option.

Example 1. *Assume a database schema with three tables that contain sensitive information about patients in hospitals in New York:*

$$T_1 : \textit{Patients}(pid, name, hiv_status, diagnosis, treatment)$$
$$T_2 : \textit{Hospital}(hid, name, address)$$
$$T_3 : \textit{Admissions}(aid, pid, hid, date)$$

A doctor with minimal database experience wants to generate an SQL query for the question $\mathcal{Q}$: "How many patients did the New York Hospital admit with HIV status as positive?". The doctor sends this query along with the database schema $\mathcal{S}$ to an LLM hosted remotely to generate the corresponding SQL query $\mathcal{Y}$. Although the LLM provider lacks access to the database and cannot execute the generated query $\mathcal{Y}$, the prompt has disclosed sensitive information: the existence of a table named "Patients", a column named "hiv_status", and possible literal values, such as "positive".

In this work, we propose a third alternative: MaskSQL, a privacy-preserving text-to-SQL framework that combines the utility of LLMs with the trust guarantees of local processing using SLMs. MaskSQL achieves this through *prompt abstraction*, which systematically replaces sensitive schema elements (table names, column names, and cell values) with abstract symbols before sending a text-to-SQL prompt to a remote LLM. Upon receiving the LLM's output, the SQL query is reconstructed locally to restore the abstracted values to their original values and make the query valid and executable.

However, implementing abstraction for text-to-SQL presents unique challenges: (i) accurately identifying sensitive tokens in the NL question according to user-defined privacy policies; (ii) preserving the utility of both the NL question and database schema in the abstraction process for accurate SQL generation; and (iii) correcting errors introduced by abstraction noise. MaskSQL addresses these with a three-stage pipeline of abstraction, SQL generation, and SQL reconstruction. On a challenging subset of the BIRD benchmark, MaskSQL outperforms state-of-the-art SLM-based approaches in accuracy, while preserving the privacy of user input, unlike LLM-based approaches.

The main contributions of this work are as follows.

1. We formalize the problem of privacy-preserving text-to-SQL using prompt abstraction guided by user-defined privacy policies.
2. We introduce MaskSQL, a framework that safeguards privacy when using untrusted remote LLMs by abstracting sensitive information in the question and database schema.
3. We propose a policy-based abstraction mechanism that enables a privacy–utility tradeoff, allowing users to customize the level of abstraction according to their needs.
4. We empirically evaluate MaskSQL on 300 complex queries from the BIRD benchmark, showing that it surpasses trusted SLM-based methods in accuracy while mitigating schema and PII leakage.

2 Background

In this section, we follow [57] to formalize the text-to-SQL task and provide some requisite background on LMs. We define a database schema as a tuple $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \Phi)$ where $\mathcal{T} = \{T_1, T_2, \ldots\}$ is the set of tables, $\mathcal{C} = \{C_1, C_2, \ldots\}$ is the set of columns across all tables, and $\Phi = \{\phi_1, \phi_2, \ldots\}$ is the set of database constraints that enforce links between tables, such as primary or foreign keys. Each C_i is also associated with a set of literal values, V_i, present in the database such that $V_i \subset \text{dom}(C_i)$, where $\text{dom}(C_i)$ is the domain of column C_i. For example, V_i could represent real numbers for a

numerical column, or textual categories for a categorical column. Herein, we consider access only to the schema $\mathcal{S}$ and not the values $V_i \in \mathcal{V}$, where $\mathcal{V}$ is the set of all literal values in the database.

Text-to-SQL Task Formulation. Text-to-SQL translation aims to map an NL question to its corresponding executable SQL code with respect to a given database schema [40]. Formally, given a natural language question $\mathcal{Q}$ and a database schema $\mathcal{S}$, the task is to generate a SQL query $\mathcal{Y}$, such that $\mathcal{Y}$ is executable and accurately represents the intent of $\mathcal{Q}$. $\mathcal{Q}$ is represented as a sequence of tokens such that $\mathcal{Q} = w_1, \ldots, w_n$ for $w_i \in \mathbb{W}$, where $\mathbb{W}$ is the vocabulary of permissible tokens (we use the English language vocabulary).

Language Models. Language Models are neural models trained on large-scale corpora of natural language and structured text. They are often informally classified into two types based on the total number of parameters. LLMs have a very high parameter count (tens to hundreds of billions) which requires multiple GPUs or specialized hardware to run, and, hence, are usually accessed through a third-party service. In contrast, SLMs use fewer parameters (less than ten billion) and can typically be run on a single GPU, enabling them to be hosted locally. Due to the difference in model complexity, SLMs have reduced utility and ability to reason compared with LLMs [50].

LMs are used by providing a prompt to specify the task to be performed. Hosted LLMs require the prompt to be passed through an API to the remote infrastructure. Once this happens, the owner loses control over how their data is processed, used, or stored, which is a major concern when there is an obligation to protect that data. Compliance with relevant laws and regulations around data privacy may preclude the use of hosted LLMs due to this loss of control. Thus, there is a need for solutions that balance the performance of LLMs with the increased control and privacy that SLMs bring.

In the context of this paper, we use LMs for the generation of SQL queries from NL questions. Given question $\mathcal{Q}$ and schema $\mathcal{S}$, we form a prompt $P(\mathcal{Q}, \mathcal{S})$ which is input to an LM denoted by f_{LM}. The text-to-SQL task is represented as $\mathcal{Y} = f_{\text{LM}}(P(\mathcal{Q}, \mathcal{S}))$, where $\mathcal{Y}$ is the generated SQL query.

3 Related Work

Text-to-SQL translation has been extensively studied in both the NLP and database communities. Earlier methods relied on rule-based systems and named-entity recognition (NER) [3, 41], followed by neural approaches using LSTMs [48, 58, 51, 53] and transformers [23, 30]. More recently, prompt-based LLMs have emerged as the state-of-the-art [38, 13]. Comprehensive comparisons of these techniques are provided on benchmarks including Spider [54], BIRD [26], Spider 2.0 [22], and in recent surveys [59, 44, 10]. An alternative line of work focuses on locally served SLMs. Several methods [39, 25, 14, 43] improve SLM performance via fine-tuning. While using local SLMs eliminates the privacy concerns of sharing data with third parties, they consistently underperform on complex queries requiring stronger reasoning [43]. Our approach follows a hybrid strategy, combining SLMs and LLMs, and we evaluate it against some state-of-the-art models reported on the BIRD leaderboard.[1]

A parallel line of work emphasizes privacy-preserving LLM inference, as user prompts often contain sensitive information. Cryptographic approaches include homomorphic encryption (HE) [6] and multi-party computation (MPC) [2], which protect user data during inference but incur significant computational overhead. Differential privacy (DP) has also been applied to preserve training privacy [33, 19, 17, 34] or inference privacy using noisy embeddings [32, 11] and token substitutions [47, 55]. However, such methods either do not operate well at the scale of LLMs, or degrade model utility [52]—a critical limitation for text-to-SQL, where precise information retrieval is required. In contrast, we propose a practical privacy protection approach, leveraging local SLMs and prompt abstraction to minimize the exposure of sensitive data while preserving semantic fidelity.

Recent work has explored prompt sanitization for LLM inference privacy [7, 56, 27, 20, 8]. For example, Portcullis [56] employs NER to detect sensitive entities, while PP-TS [20] and Pr$\varepsilon\varepsilon$mpt [8] sanitize inputs by replacing PII with contextually appropriate surrogates. HaS [7] and Papillon [27] rely on fine-tuned local models for anonymization. Related to these are generalization-based methods [42, 5, 4], which replace values with general terms to obscure specifics. Unlike these general-purpose techniques, the abstraction process we apply for text-to-SQL consistently preserves the alignment between the question's logic and the database schema.

[1] https://bird-bench.github.io/

4 Problem Statement

Consider a user with limited local compute resources who, given their database schema $\mathcal{S}$ and an NL question $\mathcal{Q}$, wants to generate a SQL query $\mathcal{Y}$ that performs the task described in $\mathcal{Q}$. The user could be, for example, a financial institution or a hospital that holds sensitive information about individuals and is prohibited from sharing that information with untrusted third parties. As such, we assume that any remotely hosted LLM is untrusted [49]. Specifically, we consider a privacy policy Ψ that defines what information in $\mathcal{S}$, $\mathcal{Q}$, or the database values $\mathcal{V}$ is considered sensitive. A privacy policy is determined by the user according to their needs and may contain all or a subset of the following information.

- **Table and Column Names:** Any table or columns names from the sets of tables $\mathcal{T}$ and columns $\mathcal{C}$ defined in $\mathcal{S}$, including words or terms in $\mathcal{Q}$ that refer such tables or columns.

- **Literal Values:** Any words or terms in $\mathcal{Q}$ that might refer to any cell value from the set $\mathcal{V}$ in the database.

By default, we use the full policy Ψ_F which includes the entire schema $\mathcal{S}$ and all values $\mathcal{V}$, as sensitive. However, less restrictive policies can also be defined, depending on the user's requirements. The problem of privacy-preserving text-to-SQL is to leverage a remote LM to accurately generate an SQL query $\mathcal{Y}$ that correctly implements the intent of $\mathcal{Q}$ without exposing any information defined in the policy Ψ.

5 MaskSQL: Privacy-Preserving Text-to-SQL Generation

Our proposed approach, MaskSQL, leverages LLM capabilities while protecting sensitive information. We identify that, for a model to generate a correct SQL query $\mathcal{Y}$, the essential information in the prompt is the mapping between the terms used in the NL question $\mathcal{Q}$ and entities in the database schema $\mathcal{S}$. Specific names or values are not crucial for the structure and syntax of the generated query. Thus, table names, column names, and literal values can be abstracted away in the prompt and later restored by using a bijective mapping between the original and abstract tokens. Consider Example 1: $\mathcal{Q}$: "*How many patients did the New York Hospital admit with HIV status as positive?*". An abstracted query, "*How many T_1 did the V_1 T_3 with C_3 as V_2?*", paired with an appropriately abstracted schema retains all information needed to generate $\mathcal{Y}$.

There are multiple challenges with respect to implementing such an abstraction approach in practice. First, the tokens in $\mathcal{Q}$ must be accurately linked to the corresponding elements in $\mathcal{S}$. This is difficult because a non-technical user may enter tokens in $\mathcal{Q}$ that do not exactly match those used in $\mathcal{S}$. For instance, the user may write the token *admit* which must be linked to the table named *Admissions* based on the semantic context. Second, the abstracted prompt is processed by an LLM, which generates a similarly abstracted SQL query. The abstracted SQL query then needs to be reliably mapped back to its concrete form. This step needs to be performed accurately, as incorrect mapping can lead to syntax errors (e.g. joins applied with incorrect column names). Third, $\mathcal{Y}$ may contain minor errors or issues (e.g., using the string value of *positive* instead of the numeric value of 1) that must be corrected. These errors can occur due to noisy translation during the process and must be corrected for error-free query execution.

MaskSQL addresses these challenges within a three-stage pipeline: *Abstraction*, *SQL Generation*, and *SQL Reconstruction*. Altogether, this pipeline safeguards user privacy while leveraging the capabilities of LLMs. An overview of the pipeline is shown in Figure 1. In the first stage, $\mathcal{Q}$, $\mathcal{S}$, and Ψ are passed to the *Abstraction* engine which produces abstracted representations $\mathcal{Q}'$ and $\mathcal{S}'$. In the second stage, these abstracted inputs are combined into a prompt $P(\mathcal{Q}', \mathcal{S}')$ and provided to the *SQL Generation* engine, which prompts a remotely hosted LLM and returns an abstracted SQL query $\mathcal{Y}'$. In the final stage, the *SQL Reconstruction* engine maps $\mathcal{Y}'$ back to its concrete form $\mathcal{Y}$, applies a final round of corrections to fix any remaining errors, and outputs the executable SQL query. Figure 4 in Appendix E shows $\mathcal{Q}'$, $\mathcal{S}'$, and $\mathcal{Y}'$ under the full policy, Ψ_F, for Example 1. The remainder of this section describes each stage in detail.

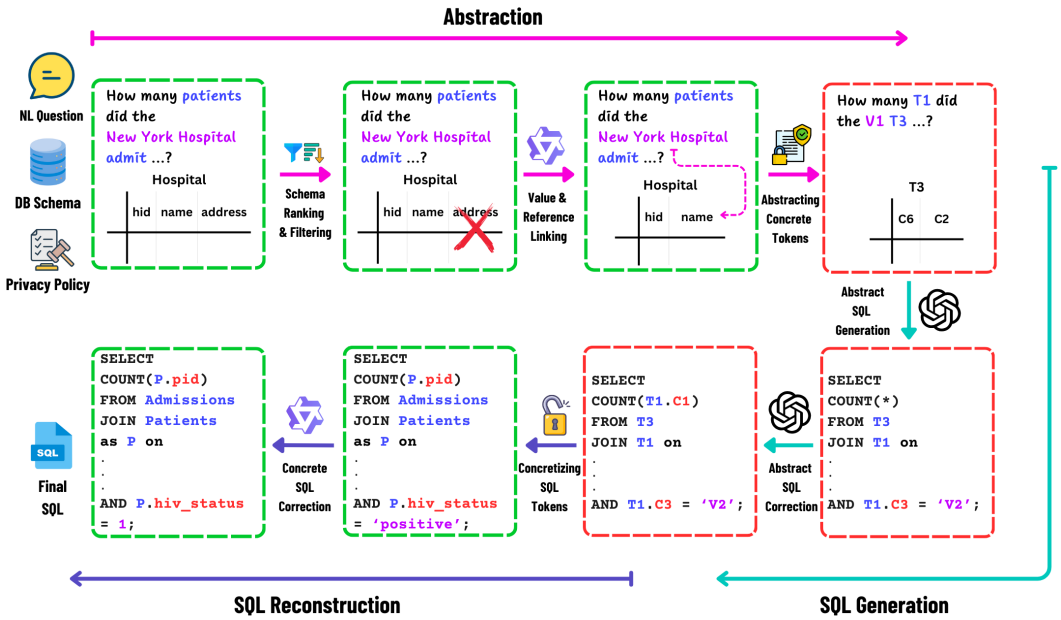

Figure 1: MaskSQL pipeline. Green dashed boxes delineate text and schema information contained in the *"trusted environment"*, while red boxes denote those exposed to *"untrusted third parties"*.

5.1 Abstraction

The first stage of the pipeline generates abstract versions of both $\mathcal{Q}$ and $\mathcal{S}$. This process consists of three main steps.

1. Ranking and filtering entire elements of $\mathcal{S}$ to drop irrelevant tables and columns based on $\mathcal{Q}$.
2. Identifying mappings between sensitive terms in $\mathcal{Q}$ and the retained schema elements using a locally hosted SLM.
3. Replacing the identified terms in $\mathcal{Q}$ and the retained schema elements with abstract identifiers.

The bijective masking procedure, combined with SLM-based schema linking, provides an effective approach for abstracting a text-to-SQL query. We detail each of these steps below.

Schema Ranking and Filtering. Real-world databases often include hundreds of tables and columns, which can overwhelm LMs with irrelevant context. To mitigate this, we use a local cross-encoder model, following the methodology of RESDSQL [24], to rank and filter schema elements based on their relevance to $\mathcal{Q}$. Specifically, RESDSQL employs a RoBERTa-based cross-encoder [28] to compute contextual embeddings for both $\mathcal{Q}$ and $\mathcal{S}$. These embeddings are pooled using a Bidirectional Long Short-Term Memory [16] and scored via Multi-Layer Perceptron modules to estimate the relevance of each table and column. Based on these scores, the top-k tables and their corresponding top-j columns are retained. This strategy is integrated into the pipeline, as keeping too few elements may result in missing relevant matches, while too many can introduce noise and increase computational overhead. As shown in Appendix C, schema filtering has a significant impact on both accuracy and efficiency. In our experiments, retaining $k = 4$ tables and $j = 5$ columns per table yields strong performance, though these parameters can be adjusted based on specific use cases. The output of this step is a ranked and filtered list of relevant schema elements, which are then passed to subsequent stages.

Value and Reference Linking. This step takes as input the filtered list of schema elements from the previous step and $\mathcal{Q}$, and constructs a mapping between the two using a local SLM. This process is performed in three sub-steps. First, the SLM is prompted to identify tokens $w_i \in \mathcal{Q}$ that may correspond to values in the database, $\mathcal{V}$. In Example 1, the tokens *New York Hospital* and *positive* are identified. In the second step, the SLM is prompted to map the identified values to their corresponding columns and tables, from $\mathcal{C}$ and $\mathcal{T}$ in $\mathcal{S}$, respectively. For Example 1, *New York Hospital* is mapped to *Hospital.name* and *positive* is mapped to *Patients.hiv_status*. In the final step, the SLM is prompted

to identify any remaining tokens $w_i \in \mathcal{Q}$ that may reference any column or table names in the filtered schema elements. In Example 1, tokens *patients*, *admit*, and *HIV status* are mapped to the tables *Patients* and *Admissions*, and column *Patients.hiv_status*, respectively. These mappings are then passed to the next step.

The importance of value and reference linking is two-fold. To protect privacy and abstract away sensitive tokens, it is essential to generate an accurate and complete linking; any token not identified as a value or reference remains unmasked and, therefore, may be exposed. In addition, the linking map must be generated precisely to preserve reference information during the abstraction process. Also, prior work shows that SLMs achieve accuracy comparable to LLMs for schema linking tasks [31], making them sufficient for this stage.

Abstracting Concrete Tokens. The final step generates the abstracted NL question $\mathcal{Q}'$ and the schema $\mathcal{S}'$. To generate $\mathcal{S}'$, each table, column, and value specified in Ψ and retained after the initial schema filtering step is assigned an abstract symbol. For example, table names are mapped to symbols like T_i, and columns to C_i. Tokens that are not included in the privacy policy Ψ remain unchanged. The mapping from $\mathcal{S}$ to $\mathcal{S}'$ is stored as a symbol lookup table, which is later used for both masking and reconstruction. Next, using this symbol table and the linking map from the previous step, all references to tables and columns in $\mathcal{Q}$ are replaced with their corresponding abstract symbols. In Example 1, the table name *Patients* is represented by the symbol T_1, column *Patients.pid* by symbol C_1, and column *Patients.name* by symbol C_2, and so on.

Finally, each literal value in $\mathcal{Q}$ is also replaced with a unique abstract symbol of the form V_i. As shown in Figure 1, the token *New York Hospital* is mapped to V_1. An additional sentence is also appended to the question to specify the column associated with the value. In Example, 1, the sentence: "V_1 is a value of the column C_7." where C_7 is the abstracted symbol for column *Hospital.name*, is appended to the abstract question. This additional context helps the LLM understand the alignment between values and schema elements without exposing any concrete tokens.

A key point is that, once the schema linking map is generated, abstraction reduces to simple text substitution. This method preserves the reference information in the question while allowing for accurate abstraction inversion of an LLM's output.

5.2 SQL Generation

Using the abstracted question $\mathcal{Q}'$ and database schema $\mathcal{S}'$ produced in the previous steps, a remotely hosted LLM is prompted to generate the corresponding abstracted SQL query $\mathcal{Y}'$. The generation prompt used in the experiments is provided in Appendix F.1. This prompt contains only $\mathcal{Q}'$ and $\mathcal{S}'$, with no additional sensitive information. Since the LLM only sees abstract symbols, the generated SQL query is also expected to follow the abstract form. An example of such an abstract SQL query is shown in Figure 1. To address minor errors in the generated abstract SQL, a self-correction mechanism, commonly used in text-to-SQL pipelines, is employed [38]. In this step, the LLM is prompted with $\mathcal{Q}'$, $\mathcal{S}'$, and $\mathcal{Y}'$ and is instructed to identify and correct any potential issues. The prompt used for this step is included in Appendix F.2.

5.3 SQL Reconstruction

In the final stage, the abstract SQL query $\mathcal{Y}'$ is mapped back to its concrete form $\mathcal{Y}$. Using the symbol lookup table created during abstraction, all abstract symbols are replaced with their corresponding concrete values. The result is an executable SQL query free of abstract identifiers. To further improve accuracy, an additional self-correction step is applied using a local SLM. The concretized SQL query is executed on the target database, and its execution result, along with $\mathcal{Q}$, $\mathcal{S}$, and the concretized query $\mathcal{Y}$, is passed to the SLM to correct any remaining errors. The full prompt for this step is provided in Appendix F.3.

6 Experiments

In this section, experimental results are reported comparing MaskSQL with several state-of-the-art text-to-SQL frameworks. We use the BIRD dataset, a widely used benchmark consisting of NL questions paired with ground-truth SQL queries, along with the corresponding databases that enable

execution-based evaluation of generated queries [26]. To demonstrate the gap between state-of-the-art LLM- and SLM-based text-to-SQL models on more complex queries, a challenging subset of 300 entries is selected from the BIRD development split. In particular, we selected examples that involve complex SQL patterns, such as nested queries, set operations (e.g., INTERSECT), and multiple joins. To generate ground-truth data for privacy measurements, GPT-4.1 [37] is used to annotate tokens that should be abstracted, followed by human review and correction. For MaskSQL, Qwen-2.5-7B-Instruct [46] serves as the trusted local SLM, and GPT-4.1 is used as the untrusted LLM for generating SQL queries from abstracted prompts.

We evaluate MaskSQL under two privacy policies.

- Ψ_F: The full privacy policy, defined in Section 4, where the entire database schema and associated values are considered sensitive.
- Ψ_C: A category-based policy where only tokens related to concepts of person names, occupations, and locations are abstracted. The formal definition is provided in Appendix A.

6.1 Baselines

We compare MaskSQL against several baselines, including three state-of-the-art text-to-SQL frameworks and direct LM prompting. The direct prompting baseline applies a simple few-shot prompt that asks the LM to generate a SQL query based on the given NL question and database schema. DAIL-SQL [13], which holds the top position on the Spider benchmark [54] for open-source solutions at the time of writing, builds on few-shot prompting with additional strategies to improve SQL generation accuracy. DIN-SQL [38], which holds the second-best ranking among open-source frameworks, decomposes text-to-SQL translation into smaller tasks to improve the accuracy. For direct prompting, DAIL-SQL, and DIN-SQL, we consider both a trusted setting, where Qwen2.5-7B-Instruct is used as the backbone LM and an untrusted setting, where GPT-4.1 is used. Finally, MSc-SQL [14] is designed specifically for SLM-only settings. It samples a few candidate SQL queries from different models and then selects the best candidate by prompting additional SLMs. For experiments leveraging MSc-SQL, we use the fine-tuned SLMs released by the authors based on Gemma-2-9B-it [45], Llama-3-8B [15], and Mistral-7B-Instruct-v0.2 [1].

6.2 Metrics

We evaluate each baseline on utility, efficiency, and privacy metrics. For utility evaluation, we use execution accuracy, as defined in the BIRD benchmark. Efficiency is measured by average token usage per query generation. For privacy, we define two metrics.

Masking Recall (MR): This metric is defined as the ratio of correctly abstracted tokens to the total number of ground-truth sensitive tokens in the NL question $\mathcal{Q}$. Higher values indicate more protection of the sensitive tokens, resulting in better privacy.

Re-identification Score (RI): This metric captures the proportion of abstracted tokens in the NL question $\mathcal{Q}'$ that cannot be re-identified by an adversary. Specifically, we prompt GPT-4.1 with the abstracted NL question $\mathcal{Q}'$ and schema $\mathcal{S}'$ and instruct it to infer the original tokens. The score is then computed as the ratio of the tokens that cannot be recovered by the LLM to the total number of abstract tokens in $\mathcal{Q}'$.

Formal definitions of these metrics are provided in Appendix B.

6.3 Results

Accuracy. Table 1 shows the execution accuracy across baselines. Untrusted LLM-based approaches perform significantly better than trusted SLM-based methods, which is expected given the additional capacity of LLMs. However, the direct use of LLMs exposes sensitive data to privacy risks. MaskSQL outperforms all trusted SLM-only approaches, achieving an execution accuracy of 55.66% under the full privacy policy, Ψ_F, which is 5 percentage points higher than the next best trusted baseline, DIN-SQL. This improvement is primarily because MaskSQL leverages LLMs in a privacy-preserving manner. However, there remains a $\sim$20 point gap between MaskSQL and the best-performing untrusted baseline. We also observe in the experiments that the direct prompting GPT-4.1 outperforms

Framework	Execution Accuracy	Token Usage	Trusted
Direct Prompting + Qwen-2.5-7B-Instruct	34.33%	1,380	Yes
DAIL-SQL + Qwen-2.5-7B-Instruct	44.33%	3,492	Yes
Fine-Tuned MSc-SQL	48.33%	8,342	Yes
DIN-SQL + Qwen-2.5-7B-Instruct	50.66%	24,812	Yes
MaskSQL (Ψ_F)	**55.66%**	**6,114**	**Yes**
MaskSQL (Ψ_C)	**62.66%**	**6,757**	**Yes**
DAIL-SQL + GPT-4.1	63.33%	3,385	No
DIN-SQL + GPT-4.1	73.66%	23,036	No
Direct Prompting + GPT-4.1	75.66%	1,352	No

Table 1: Execution accuracy and token usage of MaskSQL compared to other text-to-SQL frameworks on a subset of the BIRD development set. MaskSQL outperforms SLM-based frameworks and achieves better token efficiency than MSc-SQL and DIN-SQL. By adopting a more permissive privacy policy, MaskSQL (Ψ_C) improves execution accuracy while still preserving privacy. Pure LLM-based methods have the best accuracy, but do not protect sensitive data.

both DIN-SQL and DAIL-SQL. This is a surprising result, as these methods are supposed to enhance simple LLM prompting and yield higher accuracy. We discuss this further in Appendix D.

Efficiency. Table 1 also reports the average token usage of each framework per SQL generation. For frameworks that make more than one LM call, we compute the total token usage across all calls. As shown in the table, direct prompting with Qwen2.5 and GPT-4.1 exhibits the lowest token usage, followed by DAIL-SQL and MSc-SQL, while DIN-SQL consumes the most. MaskSQL uses a competitive number of tokens compared to SLM-based frameworks, especially when considering execution accuracy gains. For instance, it requires a quarter of the tokens consumed by the next best trusted model. The additional token usage in MaskSQL primarily comes from the intermediate linking steps and error corrections.

Privacy. Figure 2 presents the privacy scores of MaskSQL measured by the masking recall and re-identification score metrics. In Figure 2, *ground-truth masking* refers to abstracting all ground-truth tokens, providing an upper bound for masking recall and re-identification scores. Under the full policy, Ψ_F, MaskSQL achieves a masking recall of 61.36%, demonstrating that a large portion of sensitive tokens are effectively abstracted. The re-identification score measures robustness against adversarial recovery of the abstract tokens. With GPT-4.1 acting as the attacker, MaskSQL achieves a re-identification score of 75.47%, indicating that three-quarters of abstracted tokens could not be inferred from context. With ground-truth masking, this score is $\sim$86%. The relatively small gap highlights that MaskSQL is robust to contextual information leakage despite not abstracting all tokens.

We further analyze the impact of using a more permissive privacy policy. As shown in Table 1, using the category-based policy, Ψ_C, increases execution accuracy by 7 points compared to Ψ_F at the cost of a 27 point drop in masking recall, which is expected, since fewer tokens are abstracted when only tokens related to specific concepts (name, location, and occupation) are considered. This highlights the flexibility of policy-based abstraction in trading off privacy for higher accuracy, depending on the user's needs. Note that for MaskSQL under the Ψ_C setup, masking recall is computed as the ratio of correctly abstracted tokens in Ψ_C to all ground-truth sensitive tokens, rather than restricting the ground-truth tokens to only those included in the policy. Additionally, the re-identification score decreases by only 4 points, indicating that while fewer tokens are abstracted overall, the remaining abstracted tokens remain difficult for the adversary to recover.

7 Conclusion

In this work, we introduced MaskSQL, a framework for text-to-SQL translation that employs an abstraction-based privacy mechanism to preserve sensitive information according to user-defined policies when interacting with LLMs. Rather than relying solely on either SLMs or LLMs, MaskSQL follows a hybrid design that combines private local processing with SLMs and leverages the reasoning

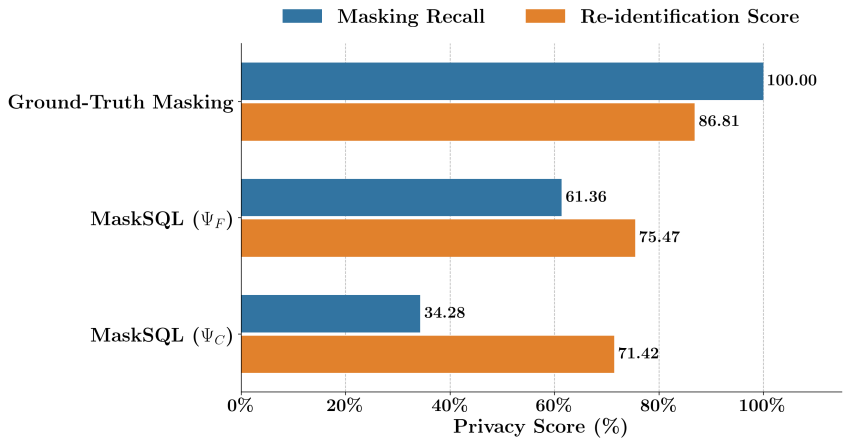

Figure 2: Privacy metrics of MaskSQL compared to ground-truth masking. Higher values indicate stronger privacy preservation.

ability of LLMs. By supporting flexible privacy policies, MaskSQL also enables a privacy-utility trade-off that allows users to tailor abstraction according to their needs. The experiments show that MaskSQL outperforms state-of-the-art SLM-based methods in terms of execution accuracy. Additionally, we implemented a category-based privacy policy that considers only sensitive tokens associated with specific semantic categories to be abstracted. The experimental results demonstrate that this more lenient policy achieves higher execution accuracy than the full privacy policy, highlighting the potential of controlled trade-offs between privacy and utility.

There are several promising directions for future work to be explored. First, since only a subset of tokens is abstracted within the NL question, the surrounding context may be exploited to re-identify the abstract tokens. Currently, we quantify this privacy risk by measuring the re-identification score. However, additional privacy measures can be applied to mitigate such potential leakage. Next, while MaskSQL uses an LLM backbone for SQL generation, its accuracy still falls behind other LLM-based frameworks. Future work will explore fine-tuning SLMs, improved schema linking, and token-efficient strategies to reduce this gap. Further, we plan to consider integrating provable privacy mechanisms, such as differential privacy, into the MaskSQL pipeline. In addition to the privacy measurements leveraged in this work, provable guarantees enhance the user's trust in the system. Finally, the abstraction-based approach in MaskSQL is not limited to the text-to-SQL translation task. We envision the abstraction approach to be applicable to tasks such as code generation, debugging, and data analysis, where reference and relational information are sufficient to perform the task. Characterizing the class of tasks to which this approach applies will be the subject of future work.

References

[1] Mistral AI. Mistral-7B-Instruct-v0.2 model card. `https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.2`, 2023.

[2] Yoshimasa Akimoto, Kazuto Fukuchi, Youhei Akimoto, and Jun Sakuma. Privformer: Privacy-preserving transformer with MPC. In *8th European Symposium on Security and Privacy*, pages 392–410. IEEE, 2023.

[3] Christopher Baik, Zhongjun Jin, Michael Cafarella, and H. V. Jagadish. Duoquest: A Dual-Specification System for Expressive SQL Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, page 2319–2329, 2020. ISBN 9781450367356. doi: 10.1145/3318464.3389776.

[4] R.J. Bayardo and Rakesh Agrawal. Data privacy through optimal k-anonymization. In *21st International Conference on Data Engineering*, pages 217–228, 2005. doi: 10.1109/ICDE.2005.42.

[5] Antorweep Chakravorty, Chunming Rong, K.R. Jayaram, and Shu Tao. Scalable, Efficient Anonymization with INCOGNITO - Framework & Algorithm. In *2017 IEEE International Congress on Big Data*, 2017. doi: 10.1109/BigDataCongress.2017.15.

[6] Tianyu Chen, Hangbo Bao, Shaohan Huang, Li Dong, Binxing Jiao, Daxin Jiang, Haoyi Zhou, Jianxin Li, and Furu Wei. THE-X: Privacy-Preserving Transformer Inference with Homomorphic Encryption. In *Findings of the Association for Computational Linguistics: ACL 2022*, 2022. doi: 10.18653/v1/2022.findings-acl.277.

[7] Yu Chen, Tingxin Li, Huiming Liu, and Yang Yu. Hide and seek (HaS): A lightweight framework for prompt privacy protection. *arXiv:2309.03057*, 2023.

[8] Amrita Roy Chowdhury, David Glukhov, Divyam Anshumaan, Prasad Chalasani, Nicolas Papernot, Somesh Jha, and Mihir Bellare. Pr$\varepsilon\varepsilon$mpt: Sanitizing Sensitive Prompts for LLMs. *arXiv:2504.05147*, 2025.

[9] Soumi Das, Camila Kolling, Mohammad Aflah Khan, Mahsa Amani, Bishwamittra Ghosh, Qinyuan Wu, Till Speicher, and Krishna P Gummadi. Revisiting Privacy, Utility, and Efficiency Trade-offs when Fine-Tuning Large Language Models. *arXiv:2502.13313*, 2025.

[10] Naihao Deng, Yulong Chen, and Yue Zhang. Recent Advances in Text-to-SQL: A Survey of What We Have and What We Expect. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 2166–2187, 2022.

[11] Minxin Du, Xiang Yue, Sherman S. M. Chow, Tianhao Wang, Chenyu Huang, and Huan Sun. DP-Forward: Fine-tuning and Inference on Language Models with Differential Privacy in Forward Pass. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, page 2665–2679, 2023. ISBN 9798400700507. doi: 10.1145/3576915.3616592.

[12] Yao Fu, Hao Peng, Litu Ou, Ashish Sabharwal, and Tushar Khot. Specializing smaller language models towards multi-step reasoning. In *International Conference on Machine Learning*, pages 10421–10430. PMLR, 2023.

[13] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. *Proc. VLDB Endow.*, 17(5):1132–1145, 2024. ISSN 2150-8097. doi: 10.14778/3641204.3641221.

[14] Satya Krishna Gorti, Ilan Gofman, Zhaoyan Liu, Jiapeng Wu, Noël Vouitsis, Guangwei Yu, Jesse C. Cresswell, and Rasa Hosseinzadeh. MSc-SQL: Multi-sample critiquing small language models for text-to-SQL translation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 2145–2160, 2025. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.107.

[15] Aaron Grattafiori et al. The Llama 3 Herd of Models. *arXiv:2407.21783*, 2024.

[16] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602–610, 2005. ISSN 0893-6080. doi: https://doi.org/10.1016/j.neunet.2005.06.042.

[17] Charlie Hou, Akshat Shrivastava, Hongyuan Zhan, Rylan Conway, Trang Le, Adithya Sagar, Giulia Fanti, and Daniel Lazar. PrE-Text: Training Language Models on Private Federated Data in the Age of LLMs. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pages 19043–19061, 2024.

[18] Yiming Huang, Jiyu Guo, Wenxin Mao, Cuiyun Gao, Peiyi Han, Chuanyi Liu, and Qing Ling. Exploring the landscape of text-to-sql with large language models: Progresses, challenges and opportunities. *arXiv:2505.23838*, 2025.

[19] Timour Igamberdiev and Ivan Habernal. DP-BART for privatized text rewriting under local differential privacy. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 13914–13934, 2023. doi: 10.18653/v1/2023.findings-acl.874.

[20] Zhigang Kan, Linbo Qiao, Hao Yu, Liwen Peng, Yifu Gao, and Dongsheng Li. Protecting user privacy in remote conversational systems: A privacy-preserving framework based on text sanitization. *arXiv:2306.08223*, 2023.

[21] Đorđe Klisura and Anthony Rios. Unmasking Database Vulnerabilities: Zero-Knowledge Schema Inference Attacks in Text-to-SQL Systems. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 6954–6976, 2025. ISBN 979-8-89176-195-7. doi: 10.18653/v1/2025.findings-naacl.386.

[22] Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin SU, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. Spider 2.0: Evaluating language models on real-world enterprise text-to-SQL workflows. In *The Thirteenth International Conference on Learning Representations*, 2025.

[23] Wenqiang Lei, Weixin Wang, Zhixin Ma, Tian Gan, Wei Lu, Min-Yen Kan, and Tat-Seng Chua. Re-examining the Role of Schema Linking in Text-to-SQL. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6943–6954, 2020. doi: 10.18653/v1/2020.emnlp-main.564.

[24] Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. RESDSQL: Decoupling Schema Linking and Skeleton Parsing for Text-to-SQL. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(11):13067–13075, 2023. doi: 10.1609/aaai.v37i11.26535.

[25] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. CodeS: Towards Building Open-source Language Models for Text-to-SQL. *Proc. ACM Manag. Data*, 2(3), 2024. doi: 10.1145/3654930.

[26] 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. 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, 2023.

[27] Siyan Li, Vethavikashini Chithrra Raghuram, Omar Khattab, Julia Hirschberg, and Zhou Yu. PAPILLON: Privacy Preservation from Internet-based and Local Language Model Ensembles. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3371–3390, 2025. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.173.

[28] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *arXiv:1907.11692*, 2019.

[29] Natasha Lomas. Italy orders ChatGPT blocked citing data protection concerns. *TechCrunch*, March 2023. URL `https://techcrunch.com/2023/03/31/chatgpt-blocked-italy/`.

[30] Jianqiang Ma, Zeyu Yan, Shuai Pang, Yang Zhang, and Jianping Shen. Mention Extraction and Linking for SQL Query Generation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, pages 6936–6942, 2020. doi: 10.18653/v1/2020.emnlp-main.563.

[31] Karime Maamari, Fadhil Abubaker, Daniel Jaroslawicz, and Amine Mhedhbi. The Death of Schema Linking? Text-to-SQL in the Age of Well-Reasoned Language Models. *arXiv:2408.07702*, 2024.

[32] Peihua Mai, Ran Yan, Zhe Huang, Youjia Yang, and Yan Pang. Split-and-Denoise: Protect large language model inference with local differential privacy. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pages 34281–34302, 2024.

[33] Jimit Majmudar, Christophe Dupuy, Charith Peris, Sami Smaili, Rahul Gupta, and Richard Zemel. Differentially private decoding in large language models. *arXiv:2205.13621*, 2022.

[34] H. Brendan McMahan, Daniel Ramage, Kunal Talwar, and Li Zhang. Learning Differentially Private Recurrent Language Models. In *International Conference on Learning Representations*, 2018.

[35] McMahon, Liv. Hundreds of thousands of Grok chats exposed in Google results. *BBC News*, August 2025. URL `https://www.bbc.com/news/articles/cdrkmk00jy0o`.

[36] Shervin Minaee, Nal Kalchbrenner, Erik Cambria, Narjes Nikzad, Meysam Chenaghlu, and Jianfeng Gao. Deep Learning–based Text Classification: A Comprehensive Review. *ACM Comput. Surv.*, 54(3), 2021. ISSN 0360-0300. doi: 10.1145/3439726.

[37] OpenAI. GPT-4.1. `https://openai.com/index/gpt-4-1/`, 2025.

[38] Mohammadreza Pourreza and Davood Rafiei. 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, 2023.

[39] Mohammadreza Pourreza and Davood Rafiei. DTS-SQL: Decomposed Text-to-SQL with Small Large Language Models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 8212–8220, 2024. doi: 10.18653/v1/2024.findings-emnlp.481.

[40] Bowen Qin, Binyuan Hui, Lihan Wang, Min Yang, Jinyang Li, Binhua Li, Ruiying Geng, Rongyu Cao, Jian Sun, Luo Si, et al. A Survey on Text-to-SQL Parsing: Concepts, Methods, and Future Directions. *arXiv:2208.13629*, 2022.

[41] Abdul Quamar, Vasilis Efthymiou, Chuan Lei, and Fatma Özcan. Natural language interfaces to data. *Foundations and Trends® in Databases*, 11(4):319–414, 2022. ISSN 1931-7883. doi: 10.1561/1900000078.

[42] Pierangela Samarati and Latanya Sweeney. Protecting Privacy when Disclosing Information: k-anonymity and Its Enforcement through Generalization and Suppression. 1998.

[43] Lei Sheng and Shuai-Shuai Xu. SLM-SQL: An Exploration of Small Language Models for Text-to-SQL. *arXiv:2507.22478*, 2025.

[44] Liang Shi, Zhengju Tang, Nan Zhang, Xiaotong Zhang, and Zhi Yang. A Survey on Employing Large Language Models for Text-to-SQL Tasks. *ACM Comput. Surv.*, 2025. ISSN 0360-0300. doi: 10.1145/3737873.

[45] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, Johan Ferret, Peter Liu, Pouya Tafti, Abe Friesen, Michelle Casbon, Sabela Ramos, Ravin Kumar, Charline Le Lan, Sammy Jerome, Anton Tsitsulin, Nino Vieillard, Piotr Stanczyk, Sertan Girgin, Nikola Momchev, Matt Hoffman, Shantanu Thakoor, Jean-Bastien Grill, Behnam Neyshabur, Olivier Bachem, Alanna Walton, Aliaksei Severyn, Alicia Parrish, Aliya Ahmad, Allen Hutchison, Alvin Abdagic, Amanda Carl, Amy Shen, Andy Brock, Andy Coenen, Anthony Laforge, Antonia Paterson, Ben Bastian, Bilal Piot, Bo Wu, Brandon Royal, Charlie Chen, Chintu Kumar, Chris Perry, Chris Welty, Christopher A. Choquette-Choo, Danila Sinopalnikov, David Weinberger, Dimple Vijaykumar, Dominika Rogozińska, Dustin Herbison, Elisa Bandy, Emma Wang, Eric Noland, Erica Moreira, Evan Senter, Evgenii Eltyshev, Francesco Visin, Gabriel Rasskin, Gary Wei, Glenn Cameron, Gus Martins, Hadi Hashemi, Hanna Klimczak-Plucińska, Harleen Batra, Harsh Dhand, Ivan Nardini, Jacinda Mein, Jack Zhou, James Svensson, Jeff Stanway, Jetha Chan, Jin Peng Zhou, Joana Carrasqueira, Joana Iljazi, Jocelyn Becker, Joe Fernandez, Joost van Amersfoort, Josh Gordon, Josh Lipschultz, Josh Newlan, Ju yeong Ji, Kareem Mohamed, Kartikeya Badola, Kat Black, Katie Millican, Keelin McDonell, Kelvin Nguyen, Kiranbir Sodhia, Kish Greene, Lars Lowe Sjoesund, Lauren Usui, Laurent Sifre, Lena Heuermann, Leticia Lago, Lilly McNealus, Livio Baldini Soares, Logan Kilpatrick, Lucas Dixon, Luciano Martins, Machel Reid, Manvinder Singh, Mark Iverson, Martin Görner, Mat Velloso, Mateo Wirth, Matt Davidow, Matt Miller, Matthew Rahtz, Matthew Watson, Meg Risdal, Mehran Kazemi, Michael Moynihan, Ming Zhang, Minsuk Kahng, Minwoo Park, Mofi Rahman, Mohit Khatwani, Natalie Dao, Nenshad Bardoliwalla, Nesh Devanathan, Neta Dumai, Nilay Chauhan, Oscar Wahltinez, Pankil Botarda, Parker Barnes, Paul Barham, Paul Michel, Pengchong Jin,

Petko Georgiev, Phil Culliton, Pradeep Kuppala, Ramona Comanescu, Ramona Merhej, Reena Jana, Reza Ardeshir Rokni, Rishabh Agarwal, Ryan Mullins, Samaneh Saadat, Sara Mc Carthy, Sarah Cogan, Sarah Perrin, Sébastien M. R. Arnold, Sebastian Krause, Shengyang Dai, Shruti Garg, Shruti Sheth, Sue Ronstrom, Susan Chan, Timothy Jordan, Ting Yu, Tom Eccles, Tom Hennigan, Tomas Kocisky, Tulsee Doshi, Vihan Jain, Vikas Yadav, Vilobh Meshram, Vishal Dharmadhikari, Warren Barkley, Wei Wei, Wenming Ye, Woohyun Han, Woosuk Kwon, Xiang Xu, Zhe Shen, Zhitao Gong, Zichuan Wei, Victor Cotruta, Phoebe Kirk, Anand Rao, Minh Giang, Ludovic Peran, Tris Warkentin, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, D. Sculley, Jeanine Banks, Anca Dragan, Slav Petrov, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Sebastian Borgeaud, Noah Fiedel, Armand Joulin, Kathleen Kenealy, Robert Dadashi, and Alek Andreev. Gemma 2: Improving Open Language Models at a Practical Size. *arXiv:2408.00118*, 2024.

[46] Qwen Team, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv:2505.09388*, 2025.

[47] Meng Tong, Kejiang Chen, Jie Zhang, Yuang Qi, Weiming Zhang, Nenghai Yu, Tianwei Zhang, and Zhikun Zhang. InferDPT: Privacy-preserving Inference for Black-box Large Language Models. *IEEE Transactions on Dependable and Secure Computing*, pages 1–16, 2025. doi: 10.1109/TDSC.2025.3550389.

[48] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7567–7578, 2020. doi: 10.18653/v1/2020.acl-main.677.

[49] Shang Wang, Tianqing Zhu, Bo Liu, Ming Ding, Xu Guo, Dayong Ye, Wanlei Zhou, and Philip S Yu. Unique security and privacy threats of large language model: A comprehensive survey. *arXiv:2406.07973*, 2024.

[50] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv:2206.07682*, 2022.

[51] Xiaojun Xu, Chang Liu, and Dawn Song. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv:1711.04436*, 2017.

[52] Biwei Yan, Kun Li, Minghui Xu, Yueyan Dong, Yue Zhang, Zhaochun Ren, and Xiuzhen Cheng. On protecting the data privacy of Large Language Models (LLMs) and LLM agents: A literature review. *High-Confidence Computing*, 5(2), 2025. ISSN 2667-2952. doi: 10.1016/j.hcc.2025.100300.

[53] Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev. TypeSQL: Knowledge-Based Type-Aware Neural Text-to-SQL Generation. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 588–594, 2018. doi: 10.18653/v1/N18-2093.

[54] 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*, pages 3911–3921, 2018. doi: 10.18653/v1/D18-1425.

[55] Xiang Yue, Minxin Du, Tianhao Wang, Yaliang Li, Huan Sun, and Sherman S. M. Chow. Differential privacy for text analytics via natural text sanitization. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 3853–3866, 2021. doi: 10.18653/v1/2021.findings-acl.337.

[56] Jiangou Zhan, Wenhui Zhang, Zheng Zhang, Huanran Xue, Yao Zhang, and Ye Wu. Portcullis: A Scalable and Verifiable Privacy Gateway for Third-Party LLM Inference. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(1):1022–1030, 2025. doi: 10.1609/aaai.v39i1.32088.

[57] Bin Zhang, Yuxiao Ye, Guoqing Du, Xiaoru Hu, Zhishuai Li, Sun Yang, Chi Harold Liu, Rui Zhao, Ziyue Li, and Hangyu Mao. Benchmarking the Text-to-SQL Capability of Large Language Models: A Comprehensive Evaluation. *arXiv:2403.02951*, 2024.

[58] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv:1709.00103*, 2017.

[59] Xiaohu Zhu, Qian Li, Lizhen Cui, and Yongkang Liu. Large Language Model Enhanced Text-to-SQL Generation: A Survey. *arXiv:2410.06011*, 2024.

A Category-Based Policy Definition

Let Ψ_F be the full policy under which the entire schema and database values are abstracted. Let $\mathcal{L}$ be the set of semantic categories. Each category in $\mathcal{L}$ is a label for a certain concept, such as "person's name" or "locations". We define the category-based policy Ψ_C for categories $\mathcal{L}$ as

$$\Psi_C = \{w \in \Psi_F \mid l(w) \in \mathcal{L}\},$$

where $l : \mathbb{W} \to \mathcal{L} \cup \{\varepsilon\}$ is a labeling function that assigns a category from the set $\mathcal{L}$ to a token $w \in \mathbb{W}$. In the case that a token does not match any category, it is mapped to ε. We assume that the labeling function is implemented with semantic text classification [36]. Here, we use Qwen2.5-7B to classify the tokens into categories based on their semantics.

B Privacy Metrics Definitions

B.1 Re-identification score (RI)

This metric is calculated by simulating an attack on the masked question. More specifically, an LLM is prompted with the masked NL question and database schema and instructed to infer the masked tokens. Then, the percentage of the masked tokens that could not be re-identified is calculated as the score. Formally, let $\mathbb{W}$ be the set of all natural language tokens and $\mathbb{S} \subseteq \mathbb{W}$ be the set of masked symbols. Let $\mathcal{Q} = w_1, w_2, \ldots, w_n$ be an NL question where $w_i \in \mathbb{W}$ for $1 \leq i \leq n$. We define $\mathcal{Q}_s = \{w_i \mid 1 \leq i \leq n\}$ as the set of tokens used in $\mathcal{Q}$. We assume that questions do not include masked tokens, i.e., $\mathcal{Q}_s \cap \mathbb{S} = \emptyset$. For a database schema $\mathcal{S}$, we define masking functions as $f : \mathbb{W} \to \mathbb{W}$, which maps each token of a question to either a masked symbol or the token itself. Let f^{-1} denote the inverse of f, which returns the original token for a masked token. For a question $\mathcal{Q}$, let $f(\mathcal{Q}) = f(w_1), f(w_2), \ldots, f(w_n)$ be its masked version. For a question $\mathcal{Q}$ and masking function f, let $\mathcal{Q}' = f(\mathcal{Q})$ be its masked version, and G be the re-identified question generated by a simulated attack on $\mathcal{Q}'$. Let $\mathcal{Q}'_s$ and G_s be the set of tokens for $\mathcal{Q}'$ and G, respectively. We define the re-identification score as follows:

$$R_f(\mathcal{Q}'_s, G_s) = 1 - \frac{|\{w \in \mathcal{Q}_s \mid f(w) \in \mathbb{S}\} \cap G_s|}{|\mathcal{Q}'_s \cap \mathbb{S}|} \tag{1}$$

where the set builder notation is used to denote the set of original tokens in the $\mathcal{Q}_s$ that have been masked by f, and $\mathcal{Q}'_s \cap \mathbb{S}$ is the set of all masked tokens in $\mathcal{Q}'_s$. A score of 1 means none of the masked tokens could be inferred by the attacker. Here, we use exact string matching and not semantic or distance-based methods.

We calculate the re-identification score for each masked question and report the average over all questions as the score for the whole dataset.

B.2 Masking Recall (MR)

Masking recall is defined as the ratio of the masked tokens to the total number of tokens in the ground-truth masking. Formally, let f, f_g be two masking functions where f_g is the ground-truth, i.e.,

a mapping that masks all ground-truth sensitive tokens. For a question $\mathcal{Q}$ and its set of tokens $\mathcal{Q}_s$, we define the masking recall as follows:

$$M_f(\mathcal{Q}_s) = \frac{\{w \in \mathcal{Q}_s \mid f_g(w) \in \mathbb{S} \wedge f(w) \in \mathbb{S}\}}{|f_g(\mathcal{Q}_s)|} \tag{2}$$

A score of 1 indicates that the tool successfully masked all required tokens. Similarly, we calculate the average of the scores to report it for a dataset.

C Ablation Study

To quantify the contribution of each component in the MaskSQL pipeline, we conducted an ablation study by removing each stage and measuring how accuracy and privacy scores are changed. The results, presented in Table 2, demonstrate that each component plays a crucial role in the MaskSQL pipeline (Figure 1).

Stage	EX	RI	MR
Complete Pipeline	**55.66%**	75.47%	61.36%
Schema Filtering	53.66%	76.49%	58.56%
Value Detection	53.33%	82.17%	64.45%
Value Linking	52.33%	74.97%	48.64%
Abstraction	47.00%	**86.79%**	**65.45%**
LLM Correction	51.33%	75.47%	61.36%
SQL Reconstruction	34.33%	75.47%	61.36%
SLM Correction	35.33%	75.47%	61.36%

Table 2: Ablation study results showing the effect of removing each component of the MaskSQL pipeline on the final execution accuracy (EX), re-identification score (RI), and masking recall (MR).

Schema Filtering. For this experiment, we remove the schema filtering at the beginning of the pipeline. As a result, the whole database schema is always passed to the further stages. As shown in the table, removing schema linking step decreases the execution accuracy and masking recall.

Value Detection. Here, we remove the separate stage for detecting the values in the question. Removing this stage slightly decreases the execution accuracy.

Value Linking. In this experiment, we remove the separate stage for linking the values of the question. While value linking slightly decreases the execution accuracy, it has a significant negative impact on the masking recall and decreases it by $\sim 13\%$.

Abstraction. In the MaskSQL pipeline, we use separate stages for schema linking and abstraction. To measure the effect of this separation, we removed the schema linking stage and used an SLM to abstract away the question with respect to the database schema. While abstraction with SLM and without a separate schema linking step improves the privacy measures, it decreases the execution accuracy by $\sim 8\%$.

LLM Correction. By removing the abstract SQL correction stage, we see a slight decrease in execution accuracy.

SQL Reconstruction. Similar to abstraction, to measure the effect of the deterministic unmasking process, we prompt SLM to unmask the SQL query and also correct the errors in the query if needed. As shown in the table, replacing the deterministic unmasking stage with language model prompting significantly reduces the execution accuracy by $\sim 21\%$.

SLM correction. The final correction stage prompts the SLM to fix errors in the reconstructed SQL query, and, as shown in the table, it has a significant impact on the execution accuracy. Removing this correction step causes a $\sim 20\%$ drop in execution accuracy.

D Discussion of Experimental Results

Table 1 shows the execution accuracy of different frameworks. As shown in the figure, direct prompting to GPT-4.1 outperforms all evaluated models. This is a surprising result, as both DIN-SQLand DAIL-SQLare also utilizing GPT-4.1 as the backend LLM and performing additional processing and improvements to enhance accuracy. The direct method uses the prompt shown in Figure F.1, but with the concrete examples instead of the abstract ones shown in the figure. Note that the BIRD benchmark website reports DIN-SQLand DAIL-SQLsubmissions based on GPT-4, rather than the later version GPT-4.1. Moreover, our experiments use a different evaluation set. These differences explain why GPT-4.1 surpassed them in our results, even though they achieved top ranks on the official BIRD leaderboard.

E Additional Figures

```
-- NL Question:
-- List the power of superheroes with height greater than 80% of the average height
↪  of all superheroes.
-- Power of superheroes refers to power_name;
-- Height greater than 80% of the average height of all superheroes = height_cm >
↪  MULTIPLY(AVG(height_cm), 0.8);


-- Qwen2.5 Output (Incorrect):
SELECT
    [superpower].[power_name]
FROM
    [hero_power]
JOIN
    [superpower] ON [hero_power].[power_id] = [superpower].[id]
JOIN
    [superhero] ON [hero_power].[hero_id] = [superhero].[id]
GROUP BY
    [superpower].[power_name]
HAVING
    [superhero].[height_cm] > AVG([superhero].[height_cm]) * 0.8;

-- GPT-4.1 Output (Correct):
SELECT
    [superpower].[power_name]
FROM
    [hero_power]
JOIN
    [superhero] ON [hero_power].[hero_id] = [superhero].[id]
JOIN
    [superpower] ON [hero_power].[power_id] = [superpower].[id]
WHERE
    [superhero].[height_cm] > (
        0.8 * (SELECT AVG([superhero].[height_cm]) FROM [superhero])
    );
```

Figure 3: An example of an SQL query that requires advanced constructs like nested queries, which Qwen2.5-7B failed to handle properly. This example is extracted from our experiments.

F Prompt Templates

F.1 SQL Generation Prompt

```
You are a SQL generation assistant. Given
(1) NL Question: a natural-language question about a dataset and
```

NL Question ($\mathcal{Q}$)

```
How many patients did the New York
↪  Hospital admit with HIV status as
↪  positive?
```

Abstract NL Question ($\mathcal{Q}'$)

```
How many T1 did the V1 T3 with C3 as
↪  V2?; V1 is a value of the column C7;
↪  V2 is a value of the column C3
```

SQL Query ($\mathcal{Y}$)

```
SELECT count(P.pid)
FROM Patient AS P
JOIN Admission AS A ON P.pid = A.pid
JOIN Hospital AS H ON A.hid = H.hid
WHERE H.name = "New York Hospital"
AND P.hiv_status = 1
```

Abstract SQL Query ($\mathcal{Y}'$)

```
SELECT count(T1.C1)
FROM T1
JOIN T3 ON T1.C1 = T3.C10
JOIN T2 ON T3.C11 = T2.C6
WHERE  T2.C7 = 'V1'
AND T1.C2 = 'V2';
```

Database Schema ($\mathcal{S}$)

```
'Patients':
    'pid':
        primary_key: true
        type: integer
    'name': text
    'hiv_status': text
    'diagnosis': text
    'treatment': text
'Hospital':
    'hid':
        primary_key: true
        type: integer
    'name': text
    'address': text
'Admissions':
    'aid':
        primary_key: true
        type: integer
    'pid':
        foreign_key: 'Patients.pid'
        type: integer
    'hid':
        foreign_key: 'Hospital.hid'
        type: integer
    'date': date
```

Abstract Database Schema ($\mathcal{S}'$)

```
'T1':
    'C1':
        primary_key: true
        type: integer
    'C2': text
    'C3': text
    'C4': text
    'C5': text
'T2':
    'C6':
        primary_key: true
        type: integer
    'C7': text
    'C8': text
'T3':
    'C9':
        primary_key: true
        type: integer
    'C10':
        foreign_key: 'T1.C1'
        type: integer
    'C11':
        foreign_key: 'T2.C6'
        type: integer
    'C12': date
```

Figure 4: Abstract question, database schema, SQL query for Example 1.

```
(2) DB Schema: the database's schema expressed in YAML
produce a single SQL SELECT statement that answers the question.

Input Format:
- DB Schema: given in YAML format where top-level keys are table names;
↪  each table lists its columns and their data types.
- Column names are case-sensitive exactly as shown in the schema.
- Each column might be primary key or a foreign key.
- For foreign key columns, fully qualified name of the referenced column is
↪  given

Output Rules
- Table and column names specified in the database schema already wrapped
↪  in brackets. You should use them with the brackets.
```

```
You should not remove the brackets when using them in the SQL.
- Each reference to a table or column name should be of the form
↪  [table_name] or [table_name].[column_name].
- Output ONLY the SQL query (no extra explanation or text).
- Use fully qualified column names: table.column.
- Only reference tables/columns that exist in the provided schema.
- Do not include any comments.
- For columns names with spaces, wrap them in backticks, e.g. "WHERE `car
↪  model` = 'bar'" instead of "WHERE car model = 'bar'".

Here are some examples:

...
-----------------------------------
#### Example 2

**NL Question:**
Among the [V1] [T1], how many of them have a [C2] of [V2]? [V1] refers to
↪  [C2] = [V1].

**DB Schema:**
[T1]:
    [C1]: text
    [C2]: real
    [C3]:
        primary_key: true
        type: integer

**Output:**
`SELECT COUNT(*) FROM [T1] WHERE [T1].[C1] = [V1] AND [C2] = [V2]

-----------------------------------
...

Now, generate a SQL query for the following question and database schema:
Inputs:
NL Question: {NL_QUESTION}
DB Schema: {DB_SCHEMA}
```

F.2 Self-Correction Prompt (Abstract)

```
You are an SQL database expert tasked with debugging a SQL query.
A previous attempt to predict a SQL query given a masked NL question and DB
↪  schema did not yield the correct results in some cases.
Either due to errors in execution, or because the result returned was empty
↪  or unexpected.
Your task is to analyze the masked SQL query given the corresponding
↪  database schema and the NL question
and fix any errors in the query if they exists.
You should then provide a corrected version of the SQL query.
Note that there may be errors in how the NL question tokens were linked and
↪  masked with the database schema elements.
As a result, the masked SQL query might contain inaccuracies based on these
↪  incorrect mappings,
and part of your task is to consider these issues as well.

**Procedure:**
1. Review Database Schema:
    - Examine the database schema to understand the database structure.
```

```
    - Database schema is given in YAML format, where top-level keys are
    ↪  table names; each table lists its columns and their data types.
    - Each column might be a primary key or a foreign key.
2. Analyze Query Requirements:
    - NL Question: Consider what information the query is supposed to
    ↪  retrieve.
    - Predicted SQL Query: Review the SQL query that was previously
    ↪  predicted and might have led to an error or incorrect result.
3. Correct the Query:
    - Modify the SQL query to address the identified issues, ensuring it
    ↪  correctly fetches the requested data according to the database
    ↪  schema and query requirements.

**Output Format:**
Present your corrected query as a single line of SQL code.
Ensure there are no line breaks within the query.
Do not include any explanations, comments, or extra text.

Here are some examples:

-------------------------------
Example 1:

NL Question:
Among the [V1] [T1], how many of them have [C2] of zero? [V1] is a
↪  nationality of [C1] = [V1];

Database Schema:
'[T1]':
    '[C1]': text
    '[C2]': real
    '[C3]':
        primary_key: true
        type: integer

The predicted SQL query was:
SELECT COUNT(*) FROM [T1] WHERE [T1].[C1] = '[V1]' AND [C2] = 0

The corrected SQL query is:
SELECT COUNT(*) FROM [T1] WHERE [T1].[C1] = '[V1]' AND [C2] = 0
-------------------------------

======= Your task =======
**************************
Database schema:
{schema}
**************************
The original question is:
NL Question: {question}
The predicted SQL query: {sql}
**************************
Based on the NL question, database schema, and the previously predicted SQL
↪  query,
Analyze the query and question, and fix the SQL query if needed.
```

F.3 Self-Correction Prompt (Concrete)

```
You are an SQL database expert tasked with correcting a SQL query.
A previous attempt to run a query did not yield the correct results,
either due to errors in execution or because the result returned was empty
↪  or unexpected.
Your role is to analyze the error based on the provided database schema and
↪  the details
of the failed execution, and then provide a corrected version of the SQL
↪  query.

**Procedure:**
1. Review Database Schema:
    - Examine the database schema to understand the database structure.
    - Iterate through each column and table name in the schema to make sure
    ↪  that it is correct.
    - Some table or column names may have white space; you should not
    ↪  change these and use a different name.
    - Database schema is given in YAML format, where top-level keys are
    ↪  table names; each table lists its columns and their data types.
    - Column names are case-sensitive, exactly as shown in the schema.
    - Each column might be a primary key or a foreign key.
    - For foreign key columns, the fully qualified name of the referenced
    ↪  column is given
2. Analyze Query Requirements:
    - Original Question: Consider what information the query is supposed to
    ↪  retrieve.
    - Executed SQL Query: Review the SQL query that was previously executed
    ↪  and led to an error or incorrect result.
    - Execution Result: Analyze the outcome of the executed query to
    ↪  identify why it failed (e.g., syntax errors, incorrect column
    ↪  references, logical mistakes).
3. Correct the Query:
    - Modify the SQL query to address the identified issues, ensuring it
    ↪  correctly fetches the requested data according to the database
    ↪  schema and query requirements.
    - For column names with spaces, wrap them in backticks, e.g., "WHERE
    ↪  `car model` = 'bar'" instead of "WHERE car model = 'bar'".

**Output Format:**
Present your corrected query as a single line of SQL code.
Ensure there are no line breaks within the query.
Do not include any explanations, comments, or extra text.

Here are some examples:

--------------------------------
Example 1:
Question:
Among the German customers, how many of the them has credit limit of zero?
↪  German is a nationality of country = 'Germany'; CREDITLIMIT = 0

Database Schema:
'[customers]':
    '[country]': text
    '[creditlimit]': real
    '[customernumber]':
        primary_key: true
        type: integer
```

```
The SQL query executed was:
SELECT COUNT(*) FROM [customers] WHERE [customers].[country] = 'German' AND
↪  [creditlimit] = 0

Output:
SELECT COUNT(*) FROM [customers] WHERE [customers].[country] = 'Germany'
↪  AND [creditlimit] = 0

The execution result:
[]
-------------------------------

======= Your task =======
**************************
Database schema:
{schema}
**************************
The original question is:
Question: {question}
The SQL query executed was: {sql}
The execution result: {exec_res}
**************************
Based on the question, table schema, and the previous query, analyze the
↪  result try to fix the query.
```