Valid Text-to-SQL Generation with Unification-based DeepStochLog

Ying Jiao[1[0009−0009−2279−7691]], Luc De Raedt[1,2[0000−0002−6860−6303]], and Giuseppe Marra[1[0000−0001−5940−9562]]

[1] KU Leuven, Dept. of Computer Science; Leuven.AI, B-3000 Leuven, Belgium
{firstname.lastname}@kuleuven.be
[2] AASS, Örebro University, Sweden

Abstract. Large language models have been used to translate natural language questions to SQL queries. Without hard constraints on syntax and database schema, they occasionally produce invalid queries that are not executable. These failures limit the usage of these systems in real-life scenarios. We propose a neurosymbolic framework that imposes SQL syntax and schema constraints with unification-based definite clause grammars and thus guarantees the generation of valid queries. Our framework also builds a bi-directional interface to language models to leverage their natural language understanding abilities. The evaluation results on a subset of SQL grammars show that all our output queries are valid. This work is the first step towards extending language models with unification-based grammars. We demonstrate this extension enhances the validity, execution accuracy, and ground truth alignment of the underlying language model by a large margin. Our code is available at https://github.com/ML-KULeuven/deepstochlog-lm.

Keywords: Generative neurosymbolic · Language models · DeepStochLog · Text-to-SQL

1 Introduction

The text-to-SQL task is to map natural language sentences to SQL queries given database schema. It provides a natural language interface to empower users regardless of their technical background to access and derive value from vast relational databases. This task also plays a central role in emerging retrieval-augmented agents [11,3,15] for various applications, such as question answering, personal assistance, and intelligent customer service. While most existing studies focus on the accuracy of queries only, we emphasize the importance of validity, ensuring that they are executable. Invalid queries that fail to execute potentially introduce vulnerabilities to automatic agents.

Deep learning models, from early recurrent ones [29,14] to recent large language models [7,18], have been successful in text-to-SQL. Though often effective, they can produce queries that violate SQL syntax and schema. Therefore, sketch-[29,32,33,4] and grammar-based approaches [31,8,14,25] guided by context-free

Table 1. A comparison of our framework and the existing approaches. Syntax and schema information suggests the guidance of syntax and schema rules. Validity guarantee underscores the assurance that the output SQL queries are always executable. Learning and inference show if the methods can be applied during the learning and inference stages.

	Syntax information	Schema information	Validity guarantee	Learning	Inference
Neural-based : [7], [18]				-	-
Sketch-based : [29], [32], [33], [4]	✓			✓	✓
Grammar-based : [31], [8], [14], [25]	✓			✓	✓
Constraint-based : [20], [13], [17]	✓	✓			✓
Execution-guided : [26], [22], [12], [6]	✓	✓			✓
Ours	✓	✓	✓	✓	✓

grammars have been proposed to avoid syntax errors. This idea is extended by constraint-based [20,13,17] and execution-guided methods [26,22,12,6] by adding schema information. They filter errors at inference time but cannot be used at learning time. These methods cannot ensure the production of valid outputs as they can exit without finding a valid query in their search space. Table 1 summarizes the properties of the studies mentioned above.

We present a neurosymbolic framework for text-to-SQL with a validity guarantee. Our framework uses DeepStochLog [27], a sequence-based neural stochastic logic programming method, as a backbone. DeepStochLog introduces neural definite clause grammars (NDCGs) which integrate stochastic definite clause grammars and neural networks. Unlike grammars employed in previous works, the unification-based, Turing-complete definite clause grammars we use can represent any syntax and schema knowledge. Our generated queries are guaranteed to have no syntax or schema errors and are always valid. To apply DeepStochLog to the text-to-SQL task, we define LM definite clause grammars (LMDCGs), an extension of NDCGs for language models. They help harness the powerful language understanding capabilities of language models and handle dynamic variable domains. In experiments, we demonstrate the effectiveness of our approach in generating valid queries on a subset of SQL grammars. Our framework also substantially improves the alignment with ground truth queries and the execution accuracy of the underlying language model. In summary, our contributions are as follows:

- We propose a neurosymbolic framework for text-to-SQL. To the best of our knowledge, we are the first to guarantee the production of valid queries with neural unification-based grammars.
- We introduce LMDCGs, an extension of DeepStochLog that integrates language models.
- We empirically show that our neurosymbolic framework significantly improves the validity, ground truth alignment, and execution accuracy of the encapsulated language model. We surpass state-of-the-art text-to-SQL approaches in terms of validity.

– We show the text-to-SQL task as a challenge and benchmark for neurosymbolic systems.

2 Problem Formulation

Given a natural language sentence nl and the schema S of a database db, the text-to-SQL task translates nl into an SQL query q. S includes: 1) a set of tables $T = \{t_1, ..., t_N\}$ of size N, and 2) a set of columns $C = \{c_1^1, ..., c_{n_1}^1, ..., c_1^N, ..., c_{n_N}^N\}$ linked to the tables, where n_i represents the number of columns in table t_i.

Fig. 1 demonstrates the workflow of our framework. We aim to generate a valid and correct q that can retrieve the right answers to nl from db without runtime errors.

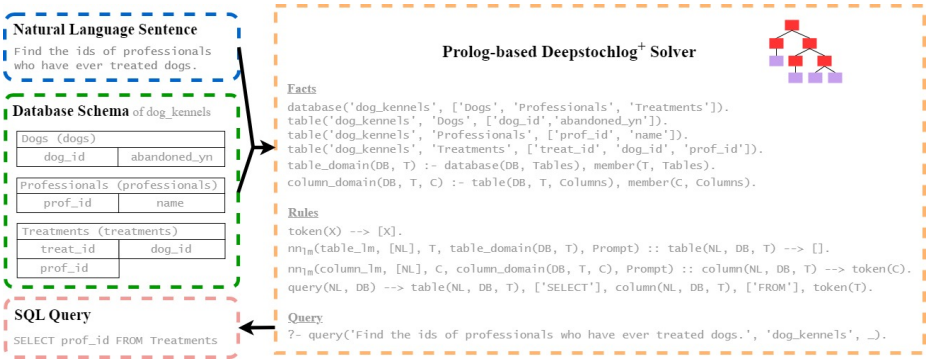

Fig. 1. Illustration of a text-to-SQL instance solved by our framework (basic grammar is used for brevity). Given the inputs, the system maximizes the probability of the ground truth SQL query when it is known and produces the most probable query when the target is unknown. The first LMDCG rule nn_{lm} in the logic program prompts the language model *table_lm* and gets a probability distribution over the three tables in the dog_kennels database. Similarly, the second one prompts *column_lm* and gets a probability distribution over the columns in a given table. The inference steps are shown in Fig. 2.

3 Preliminaries

This section provides essential background information on grammar and DeepStochLog. We refer to the original DeepStochLog paper [27] for more details.

Context-free grammars (CFGs) define a set of rewriting rules of the form $V \rightarrow W_1, ..., W_n$, where V is a non-terminal and W_i is either a terminal or a non-terminal. **Definite clause grammars (DCGs)** are a popular logic-programming-based extension of CFGs that can be executed as Prolog

programs [16]. They are unification-based and can encode context-sensitive languages. DCGs replace the non-terminals in CFGs by logical atoms $a(l_1, .., l_n)$ with a predicate a and n terms l_i. A term is a constant, a logical variable, or a structured term $f(l_1, ..., l_k)$ where f is a functor. DCG rules take the format $nt \rightarrow g_1, ..., g_n$, where nt is an atom, a goal $g_1, ..., g_n$ is a sequence and g_i is an atom or a list of terminals and logical variables. **Stochastic definite clause grammars (SDCGs)** formed as $p_i :: nt \rightarrow g_1, ..., g_n$ add probabilities p_i to DCG rules. They define a probability distribution over possible parses of a sequence and allow the most likely parse to be determined. SDCGs require the probabilities of the rules with the same non-terminal predicate to sum to 1.

DeepStochLog extends SDCGs to **neural definite clause grammars (NDCGs)** that integrate neural networks. An NDCG rule is defined as $nn(m, [I_1, ..., I_X], [O_1, ..., O_Y], [D_1, ..., D_Y]) :: nt \rightarrow g_1, ..., g_n$. The nn predicate denotes a neural network m that takes input variables $I_1, ..., I_X$ and outputs a probability distribution over variables $O_1, ..., O_Y$ with domains $D_1, ..., D_Y$. For instance, $nn(table\_nn, [\text{"Find ... dogs."}], T, [\text{"Dogs"}, \text{"Professionals"}, \text{"Treatments"}]) :: table(\text{"Find ... dogs."}) \rightarrow [T]$ represents a neural network $table\_nn$ that takes a natural language sentence "Find the ids of professionals who have ever treated dogs" as input and outputs a probability distribution over the table domain of "Dogs", "Professionals", "Treatments".

Inference in DeepStochLog (see Section 4.1) computes the probability of a logical goal given an input sequence and a DCG, using probabilities computed by neural networks. The set of parses of the input sequence is translated first into a logical proof tree (e.g. Fig. 2 (a)), which is then turned into a computational graph (e.g. Fig. 2 (b)) that computes the likelihood of the goal. In particular, the probability $P_G(derives(G, T))$ is computed, where G is a logical goal (e.g. the starting symbol) and T is a sequence to parse. Logical inference uses resolution to find all derivations $d(G\theta)$ that produce T with an answer substitution θ. The resolution process is then translated into an AND-OR circuit. Probability inference calculates $P_G(derives(G, T)) = \sum_{d(G\theta)=T} P((G\theta)) = \sum_{d(G\theta)=T} \prod_{r_i \in d(G\theta)} p_i^{k_i}$, where p_i is the probability of rule r_i used for k_i times in a derivation. This computation equals a bottom-up evaluation of the AND-OR circuit where the logical circuit is compiled to an arithmetic circuit with the $(+, \times)$ semiring [9]. Similarly, the most probable derivation for G can be identified with the $(\max, \times)$ semiring.

Learning in DeepStochLog (see Section 4.1) is cast into the maximization of the likelihood of the input sequences. It is defined as

$$\min_p \sum\nolimits_{(G_i\theta_i, T_i, t_i) \in \mathcal{D}} \mathcal{L}(P_G(derives(G_i\theta_i, T_i); p), t_i) \tag{1}$$

where p is a vector of rule probabilities, $\mathcal{D}$ is a dataset of triples $\{G_i\theta_i, T_i, t_i\}$, t_i is a target probability and $\mathcal{L}$ is a differentiable loss function. This learning problem is solved with standard gradient descent techniques like the Adam optimizer [10]. The gradients of $\mathcal{L}$ w.r.t p can be computed automatically and backpropagated seamlessly to train the internal parameters of neural networks.

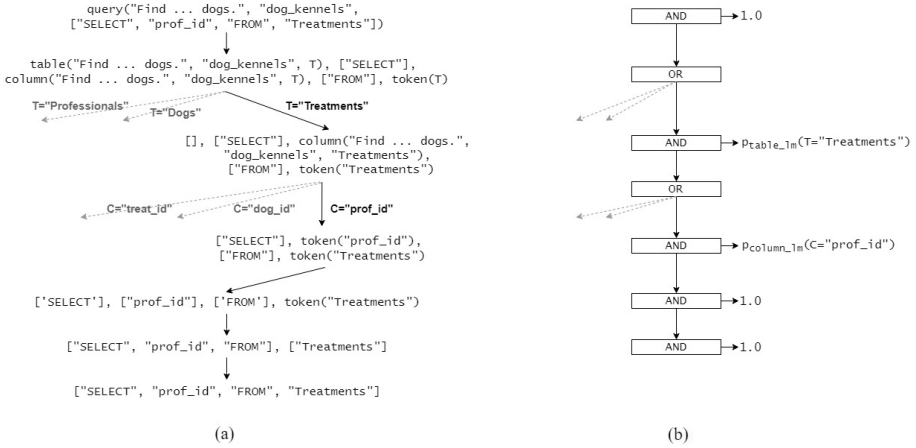

Fig. 2. Inference steps on the text-to-SQL instance in Fig. 1. (a) The SLD tree for *derives*(*query*("Find the ids of professionals who have ever treated dogs.", "dog_kennels", ["SELECT", "prof_id", "FROM", "Treatments"])). Thanks to unification, the branches of the wrong table and column substitutions will fail. Failing branches are in grey. (b) The corresponding AND-OR circuit. The probabilities of failing branches are not considered.

4 Methodology

We propose a neurosymbolic framework for text-to-SQL based on LM definite clause grammars (LMDCGs). LMDCGs are an extension of neural definite clause grammars (NDCGs) that integrate stochastic definite clause grammars (SDCGs) with language models. In Section 4.1, we show the workflow of our framework with the text-to-SQL instance in Fig. 1. We define LMDCGs and illustrate the bi-directional interface to language models they provide in Section 4.2. Lastly, we demonstrate the advantage of our definite clause grammars (DCGs) over the rules in the previous sketch- and grammar-based approaches in Section 4.3.

4.1 Our Workflow

Our logic program has three parts: facts, rules, and a Prolog query.

Facts The facts represent associations between the database, tables, and columns. They are automatically generated from the database schema to define the domain of possible table and column variable substitutions. For example, in Fig. 1, the fact with *database* predicate describes the three tables in the database "dog_kennels". The *table* predicate describes the columns in each table. The *table_domain* and *column_domain* predicates retrieve the tables in dog_kennels and the columns in a given table respectively.

Rules The rules encoding DCGs will be used to find answers to $query(nl, db, Q)$, where Q is the ground truth SQL query. Each rule can be assigned a probability. In our task, language models determine probabilities of LMDCG rules using the predicate nn_lm (see Section 4.2). Fig. 2 (a) shows how the rules are applied to parse the ground truth SQL query ["SELECT", "prof_id", "FROM", "Treatments"] step by step. As in Fig. 2 (b), $table\_lm$ and $column\_lm$ determine the probability of the table and column grammar branches respectively. The *query* and *token* rules are deterministic, i.e. purely logical, with a probability of 1.0.

Prolog Query The Prolog query[3] $query(nl, db, Q)$ defines the output of our framework. During training, Q is known and the system outputs the probability of producing Q given natural language sentence nl and database db. To this end, DeepStochLog inference with the $(+, \times)$ semiring is used (see Section 3). For example, in Fig. 2, given the sentence "Find ... dogs" and the database dog_kennel, the probability of the ground truth SQL query

$$P(derives(query(\text{"Find ... dogs."}, \text{"dog\_kennels"}, [\text{"SELECT"}, \text{"prof\_id"},$$

$$\text{"FROM"}, \text{"Treatments"}])) = 1.0 \times (0 + 0 + P_{table\_lm}(\text{T="Treatments"}) \times$$

$$(0 + 0 + P_{column\_lm}(\text{C="prof\_id"}) \times 1.0 \times 1.0))$$

The learning process maximizes the probability of this query. Since DeepStochLog produces end-to-end differentiable inference graphs (i.e. AND/OR circuit in Fig. 2 (b)), this can be easily achieved by standard backpropagation and stochastic gradient descent. During evaluation, the Q is unknown. The system outputs the most likely SQL query given nl and db. To this end, DeepStochLog inference with the $(\max, \times)$ semiring is used.

4.2 LMDCGs

We define an LMDCG rule as:

$$nn_{lm}(lm, [NL], O_Y, D_Y, Prompt) :: nt \rightarrow g_1, ..., g_n$$

where lm is a language model, NL is a natural language sentence, O_Y is an output variable with domain $D_Y = [y_1, ..., y_n]$, and $Prompt$ is a constant sentence.

LMDCGs communicate with the underlying language model bi-directionally: 1) by constructing the input to execute them; and 2) by re-normalizing their output to build probabilities for SDCGs. Since the probabilities produced by language models are used during inference, we can backpropagate gradients seamlessly to language models and fine-tune them.

[3] Notice that the Prolog query is the logical goal to be proved, which differs from the SQL query to generate.

Language Model Input Construction In the text-to-SQL task, the language model input is designed as the concatenation of 1) NL, 2) the possible substitutions of O_Y with their indexes in D_Y, i.e. "Answer i for y_i", and 3) $Prompt =$ "the answer should be Answer". The substitutions of O_Y in D_Y can be tokenized into several parts with language models, for example, the column name "abandoned_yn". We require the language models to output indexes instead of the substitutions to make them treat every substitution as a whole. For example, in Fig. 1, given the natural language sentence "Find ... dogs.", the table domain ["Dogs", "Professionals", "Treatments"], and the rule "$nn_{lm}(table\_lm, [NL], T, table\_domain(DB, T), Prompt) :: table(NL, DB, T) \rightarrow []$.", the input to the *table_lm* is "Find ... dogs. Answer 1 for Dogs, Answer 2 for Professionals, Answer 3 for Treatments, the answer should be Answer ". *table_lm* takes this input and outputs a logit for every token in its vocabulary.

Language Model Output Normalization To get the probability distribution over D_Y, for language models with a decoder, we extract the logits for indexes i from the decoder outputs and renormalize them with the softmax function. For the encoder-only language models, we apply a linear layer on top of them. For example, in Fig. 1, the logits for the token "1", "2" and "3" are extracted and the softmax distribution could be 0.2, 0.2, and 0.6. Thus, the LMDCG rule represents a set of grammar branches: $0.2 :: table($"Find ... dogs.", "dog_kennels", "Dogs"$) \rightarrow []; 0.2 :: table($"Find ... dogs.","dog_kennels", "Professionals"$) \rightarrow []; 0.6 :: table($ "Find ...dogs.", "dog_kennels", "Treatments"$) \rightarrow []$. Here, we apply the empty production. The rule produces an empty sequence but provides substitutions of the table variable T, which helps determine the column domain in the column rule through the Prolog unification mechanism. Unlike the autoregressive models, the empty production allows us to deal with tables before columns which better fits human intuitions.

4.3 DCGs v.s. Previous Rules

The rules in previous sketch- [29,32,33,4] and grammar-based approaches [31,8,14,25] do not explicitly represent relationships between tables and columns. When generating basic SQL queries, they are equivalent to CFGs in Example 1 a). Considering the dog_kennels database in Fig. 1, these approaches can overgenerate and lead to invalid queries mismatching tables and columns, for example, "SELECT treat_id FROM Dogs".

The example DCG avoids this error and encodes the associations of tables and columns by bounding the column domain to columns in a specific table. We demonstrate that DCGs can guarantee the correctness of syntax and the faithfulness to schema and thus guarantee the validity of outputs with this basic SQL generation scenario. As DCGs are Turing-complete, they can be generalized to express sophisticated SQL syntax and semantic constraints and produce valid, advanced SQL queries.

Example 1. An equivalent CFG of rules in previous works a) and a DCG b).

a) $T \rightarrow$ "Dogs", "Professionals", "Treatments"
$C \rightarrow$ "dog_id", "abandoned_yn", "prof_id", "name", "treat_id"
$Q \rightarrow$ "SELECT" C "FROM" T

b) $table$("Dogs") $\rightarrow$ "Dogs"
$table$("Professionals") $\rightarrow$ "Professionals"
$table$("Treatments") $\rightarrow$ "Treatments"
$column$("Dogs") $\rightarrow$ "dog_id", "abandoned_yn"
$column$("Professionals") $\rightarrow$ "prof_id", "name"
$column$("Treatments") $\rightarrow$ "treat_id", "dog_id", "prof_id"
$Q \rightarrow$ "SELECT" $column(T)$ "FROM" $table(T)$

5 Experiments

5.1 Research Questions

Our experiments aim to address the following questions:

Q1 Which language model produces more correct queries?
Q2 Does our framework ensure validity? How does it compare to other text-to-SQL approaches?
Q3 Does our framework improve exact matching and execution accuracy?

5.2 Tasks

Task 1 (Q1) We explore which language model should we encapsulate to achieve better performance. The language models in our framework can be classified into two types. Type 1 produces probability distributions over table and column grammar branches with dynamic domains. For type 1, we use T5-small models with an encoder-decoder structure. T5 [19] is popular in addressing the text-to-SQL task [21,20]. The vocabulary of its decoder allows us to handle the domains of the table and the column that vary with the database. Type 2 provides probability distributions over grammar branches defined by SQL syntax with fixed output domains. For example, the selection branches between "SELECT" and "FROM" could be "*", "COUNT(*)", a column with an aggregation function, etc.

In task 1, we compare T5-small with Bert-base [5] plus a linear layer for type 2 using a grammar for the SELECT clause (in Appendix A.1). Setting 1 uses two T5-small models and one Bert-base for the table, column, and selection branch respectively. Setting 2 uses three T5-small models. The training and evaluation details for task 1 are in Appendix A.2.

Task 2 (Q2, Q3) We evaluate our system with a recursion-free grammar covering the SELECT, WHERE, GROUP BY, ORDER BY clauses and EXCEPT between two simple SELECT clauses. The recursive cases are left for future studies. The exact inference in Section 3 is inefficient for the extended grammar. We use the greedy inference that takes the most likely grammar branch. Appendix B describes more training and evaluation details, the grammar, and the language models employed.

We compare our framework with the following baselines:

- Neural-based:
  - T5-small: fine-tuned T5-small that treats text-to-SQL as sequence-to-sequence generation.
  - DAIL-SQL [7] and DIN-SQL [18]: GPT-4 [1] under few-shot prompting.
- Grammar-based:
  - T5-small + CFGs: an ablation of our framework which does not consider the relations between table and column. It simulates the rules in previous sketch- [29,32,33,4] and grammar-based approaches [31,8,14,25].
- Constraint-based:
  - Graphix-T5 [13]: T5-3B augmented with graph-aware layers and constrained decoding PICARD [20].
- Execution-guided:
  - C3 [6]: zero-shot ChatGPT. The final output is selected based on execution results.

More information on the methodology of baselines is in Section 6. Their implementation details are in Appendix C

Following [34], we consider evaluation metrics: exact matching that compares the predicted and the ground truth query, and the execution accuracy that compares their execution results. We also report validity that checks whether the predicated queries are executable.

Data We extract samples that satisfy the scope of our grammar from Spider [34], a large-scale complex and cross-domain benchmark dataset for text-to-SQL. All models are evaluated on the instances extracted from Spider's development division. In task 1, we employ 384 training samples and 59 evaluation samples. Task 2 uses 2106 training samples and 258 evaluation samples.

5.3 Results

Q1 For task 1 setting 1 that uses Bert-base for the selection branch, the percentage of outputs with correct execution results is limited to 61%. Its results always start with "SELECT COUNT(*)". This is caused by the unbalanced training data shown in Table 2. Setting 2 using T5-small for the selection branch is less affected by the biased data. 93% outputs lead to the right results. Since the training data for grammar branches is often unbalanced, we encapsulate T5-small for all neural components in task 2 experiments.

Table 2. Selection branch statistics of task 1 training data. col. stands for column.

*	COUNT(*)	col.	COUNT(col.)	SUM(col.)	AVG(col.)	MIN(col.)	MAX(col.)
2.1%	55.5%	14.9%	3.7%	7.3%	12.1%	0.5%	3.9%

Q2 Table 3 compares our framework with state-of-the-art text-to-SQL approaches. Our DeepStochLog with LMDCGs is able to guarantee validity in 100% of the test queries. We ensure faithfulness to both SQL syntax and database schema. In comparison, neural-based methods (T5-small, DAIL-SQL, and DIN-SQL) without hard schema constraints can produce non-valid queries by using identifiers not defined in the schema. T5-small + CFGs can mismatch tables and columns due to the lack of constraints on their relations. Graphix with constrained decoding can exit without finding a valid query and output incomplete results. C3 with execution-guided decoding also produces 100% valid queries for the extracted evaluation examples. However, execution is not always feasible in application scenarios. Table 4 shows examples of common errors made by the baselines.

Table 3. Comparison with state-of-the-art models for text-to-SQL on the selected subset of Spider. (*) Execution-based methods are not applicable in real settings as they need to execute the query during generation. Params. means parameters.

		Validity%	Exact Matching %	Execution Accuracy%
Smaller Models (Millions Params.)	T5-small	53.9	41.1	41.1
	T5-small+CFGs	88.8	67.1	70.9
	Ours (T5-small+DCGs)	**100.0**	75.6	77.9
Larger Models (B/Trillions Params.)	DAIL-SQL (GPT-4)	99.2	88.8	89.9
	DIN-SQL (GPT-4)	99.2	78.7	90.7
	Graphix-T5 (T5-3B+PICARD)	99.6	**91.9**	**91.9**
Execution Required	C3 (ChatGPT+Execution)	**100.0***	80.6*	85.3*

Table 4. Examples of invalid outputs from baseline models.

	Example	Error
T5-small	SELECT Name FROM country WHERE Independence ¿ 1950	Invent identifiers Independence not in schema
T5-small+CFGs	SELECT COUNT(*) FROM Has_Pet WHERE weight ¿ 10	Mismatch table and column, weight not in Has_Pet
DAIL-SQL	SELECT Paragraph_Details FROM Paragraphs WHERE Paragraph_Text LIKE '%Korea%'	Invent identifiers Paragraph_Details not in schema
DIN-SQL	SELECT T1.first_name, ... ORDER BY T2.rank_points DESC LIMIT 1	Invent identifiers rank_points not in schema
Graphix	select	Incomplete query

Q3 The performance on exact matching and execution accuracy (see Table 3) also suggests the effectiveness of integrating unification-based grammars with language models. Compared to the vanilla T5-small model, our framework that extends T5-small with unification-based grammars improves the exact matching and execution accuracy by a large margin. We also outperform T5-small + CFGs due to the bounding of column domains to corresponding tables. However, T5-small, with 60 million parameters, limits the capability of our framework to produce correct SQL queries. As shown in Table 5, our outputs with incorrect execution results are caused by misunderstanding user intentions and linking the natural language sentences to the wrong SQL components. The results of DAIL-SQL, DIN-SQL, Graphix, and C3 indicate that this problem could be alleviated by accessing larger-scaled models with billions or trillions of parameters and by employing graph-aware layers that model relations better.

Table 5. Examples of incorrect outputs from our framework. Pred. refers to predication.

Mislinking Type	Example	Proportion %
Identifier(s)	Find the number of distinct name of losers.	65.0
	Pred.: SELECT COUNT(DISTINCT first_name) FROM players	
	Gold: SELECT count(DISTINCT loser_name) FROM matches	
Selection branch	Count the number of dogs that went through a treatment.	19.7
	Pred: SELECT COUNT(*) FROM Treatments	
	Gold: SELECT count(DISTINCT dog_id) FROM Treatments	
Clause	Count the number of high schoolers.	12.0
	Pred: SELECT COUNT(*) FROM Highschooler WHERE grade = 1	
	Gold: SELECT count(*) FROM Highschooler	
Operator	Which cities do more than one employee under age 30 come from?	3.3
	Pred: SELECT City FROM employee WHERE Age ¿ 30 GROUP BY City HAVING COUNT(*) ¿ 1	
	Gold: SELECT city FROM employee WHERE age ¡ 30 GROUP BY city HAVING count(*) ¿ 1	

6 Related Work

Neural-based The state-of-the-art text-to-SQL approaches are based on large language models (LLMs). They do not employ any hard constraints to guarantee valid outputs. DAIL-SQL [7] designs the few-shot prompt for LLMs. It uses code question representation and selects examples based on both question and query. DIN-SQL [18] decomposes the text-to-SQL task into sub-problems: schema linking, classification and building different prompts from each class, SQL generation, and self-correction. The first three steps are conducted by LLMs under the few-shot setting and the last one by instructing a LLM.

Sketch-based Sketch-based methods [29,32,33,4] model text-to-SQL as a sequence-to-set problem considering the possible equivalent serialization of one query.

They define a dependency graph of slots filled by independently trained neural components. The query synthesis process is viewed as an inference on the graph. Their sketches agree with SQL syntax but do not encode schema information like table and column relations. Therefore, the produced queries can be non-executable due to violations of schema.

Grammar-based Grammar-based methods [31,25,8,14] introduce a sequence-to-action formalism of text-to-SQL that generates a derivation Abstract Syntax Tree [30] or a similar intermediate representation [8]. At each time step, they use context-free grammars to define the possible actions, and a trained neural model to predict the probability distribution over the action set. Similar to sketch-based approaches, these probabilistic grammar models cannot ensure the validity of their outputs as they do not include semantic constraints.

Constraint-based and Execution-guided Constraint-based methods [20,17] accept only tokens that align with defined SQL syntax and semantic constraints during decoding. Graphix-T5 [13] applies constraint-based PICARD [20] to prune erroneous tokens during its beam-search phase. It also enhances T5 with graphix layers to better model the relational structures in text-to-SQL. Execution-guided approaches [26,22,12] execute queries and filter out faulty ones during generation. C3 [6] samples a set of SQL queries from zero-shot ChatGPT and votes for the most consistent one based on their execution results. The methods in these two categories are effective but can fail to generate valid outputs when the underlying models put low probabilities on valid predictions.

7 Conclusion and Future Work

We introduce LM definite clause grammars (LMDCGs), an extension of DeepStochLog [27] that integrates language models with unification-based grammars to provide a validity guarantee to the text-to-SQL task. We evaluate our method on a subset of SQL syntax. The results suggest that this integration eliminates non-executable queries and significantly contributes to the alignment with ground truth queries and execution accuracy.

Several limitations of our framework are interesting to explore in future studies. First, the prompt part of LMDCGs is not yet built dynamically to indicate the parsing states. Second, our system could be further enhanced by employing billion-level open-source large language models like Llama [24], Falcon [2], and Alpaca [23]. Lastly, scaling the current framework to larger grammars is not trivial. An interesting direction for speeding up the inference process would be searching for the k-best derivations.

Acknowledgments

This project has received funding from the European Union's Horizon Europe research and innovation programme under the Marie Skłodowska-Curie grant

agreement No 101073307 and the Flemish Government (AI Research Program). Luc De Raedt is also supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. We thank Thomas Winters for the helpful discussions. We also thank the anonymous reviewers for their valuable feedback.

References

1. Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F.L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al.: Gpt-4 technical report. arXiv preprint arXiv:2303.08774 (2023)
2. Almazrouei, E., Alobeidli, H., Alshamsi, A., Cappelli, A., Cojocaru, R., Debbah, M., Goffinet, E., Heslow, D., Launay, J., Malartic, Q., Noune, B., Pannier, B., Penedo, G.: Falcon-40B: an open large language model with state-of-the-art performance (2023)
3. Chase, H.: LangChain (2022), `https://github.com/langchain-ai/langchain`
4. Choi, D., Shin, M., Kim, E., Shin, D.: Ryansql: Recursively applying sketch-based slot fillings for complex text-to-sql in cross-domain databases. Computational Linguistics **47**(2), 309–332 (2021)
5. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers). pp. 4171–4186 (2019)
6. Dong, X., Zhang, C., Ge, Y., Mao, Y., Gao, Y., Lin, J., Lou, D., et al.: C3: Zero-shot text-to-sql with chatgpt. arXiv preprint arXiv:2307.07306 (2023)
7. Gao, D., Wang, H., Li, Y., Sun, X., Qian, Y., Ding, B., Zhou, J.: Text-to-sql empowered by large language models: A benchmark evaluation. arXiv preprint arXiv:2308.15363 (2023)
8. Guo, J., Zhan, Z., Gao, Y., Xiao, Y., Lou, J.G., Liu, T., Zhang, D.: Towards complex text-to-sql in cross-domain database with intermediate representation. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics. pp. 4524–4535 (2019)
9. Kimmig, A., Van den Broeck, G., De Raedt, L.: An algebraic prolog for reasoning about possible worlds. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 25, pp. 209–214 (2011)
10. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
11. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.t., Rocktäschel, T., et al.: Retrieval-augmented generation for knowledge-intensive nlp tasks. Advances in Neural Information Processing Systems **33**, 9459–9474 (2020)

12. Li, H., Zhang, J., Li, C., Chen, H.: Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 37, pp. 13067–13075 (2023)
13. Li, J., Hui, B., Cheng, R., Qin, B., Ma, C., Huo, N., Huang, F., Du, W., Si, L., Li, Y.: Graphix-t5: Mixing pre-trained transformers with graph-aware layers for text-to-sql parsing. In: Proceedings of the AAAI Conference on Artificial Intelligence. p. 13076–13084 (2023)
14. Lin, K., Bogin, B., Neumann, M., Berant, J., Gardner, M.: Grammar-based neural text-to-sql generation. arXiv preprint arXiv:1905.13326 (2019)
15. Liu, J.: LlamaIndex (2022). https://doi.org/10.5281/zenodo.1234, `https://github.com/jerryjliu/llama_index`
16. Pereira, F.C., Warren, D.H.: Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. Artificial intelligence **13**(3), 231–278 (1980)
17. Poesia, G., Polozov, O., Le, V., Tiwari, A., Soares, G., Meek, C., Gulwani, S.: Synchromesh: Reliable code generation from pre-trained language models. arXiv preprint arXiv:2201.11227 (2022)
18. Pourreza, M., Rafiei, D.: Din-sql: Decomposed in-context learning of text-to-sql with self-correction. Advances in Neural Information Processing Systems **36** (2024)
19. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J.: Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of machine learning research **21**(140), 1–67 (2020)
20. Scholak, T., Schucher, N., Bahdanau, D.: Picard: Parsing incrementally for constrained auto-regressive decoding from language models. In: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing. pp. 9895–9901 (2021)
21. Shaw, P., Chang, M.W., Pasupat, P., Toutanova, K.: Compositional generalization and natural language variation: Can a semantic parsing approach handle both? In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers). pp. 922–938 (2021)
22. Suhr, A., Chang, M.W., Shaw, P., Lee, K.: Exploring unexplored generalization challenges for cross-database semantic parsing. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. pp. 8372–8388 (2020)
23. Taori, R., Gulrajani, I., Zhang, T., Dubois, Y., Li, X., Guestrin, C., Liang, P., Hashimoto, T.B.: Stanford alpaca: An instruction-following llama model. `https://github.com/tatsu-lab/stanford_alpaca` (2023)
24. Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al.: Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971 (2023)
25. Wang, B., Shin, R., Liu, X., Polozov, O., Richardson, M.: 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. pp. 7567–7578 (2020)
26. Wang, C., Tatwawadi, K., Brockschmidt, M., Huang, P.S., Mao, Y., Polozov, O., Singh, R.: Robust text-to-sql generation with execution-guided decoding. arXiv preprint arXiv:1807.03100 (2018)
27. Winters, T., Marra, G., Manhaeve, R., De Raedt, L.: Deepstochlog: Neural stochastic logic programming. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 36, pp. 10090–10100 (2022)

28. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al.: Transformers: State-of-the-art natural language processing. In: Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations. pp. 38–45 (2020)
29. Xu, X., Liu, C., Song, D.: Sqlnet: Generating structured queries from natural language without reinforcement learning. arXiv preprint arXiv:1711.04436 (2017)
30. Yin, P., Neubig, G.: A syntactic neural model for general-purpose code generation. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 440–450 (2017)
31. Yin, P., Neubig, G.: Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing (Demo Track) (2018)
32. Yu, T., Li, Z., Zhang, Z., Zhang, R., Radev, D.: 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) (2018)
33. Yu, T., Yasunaga, M., Yang, K., Zhang, R., Wang, D., Li, Z., Radev, D.: Syntaxsqlnet: Syntax tree networks for complex and cross-domain text-to-sql task. In: Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing. pp. 1653–1663 (2018)
34. Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., et al.: 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. pp. 3911–3921 (2018)

A Task 1

A.1 Logic Program

Task 1 grammar is modeled as follows:

```
S_domain(Y) :- member(Y, [0, 1, 2, 3, 4, 5, 6, 7]).
table_domain(DB, T) :- database(DB, Tables), member(T, Tables).
column_domain(DB, T, C) :- table(DB, T, Columns), member(C, Columns).
token(X) --> [X].
selection_branch(_, _, _, 0) --> ['*'].
selection_branch(_, _, _, 1) --> ['COUNT(*)'].
selection_branch(NL, DB, T, 2) --> column(NL, DB, T).
selection_branch(NL, DB, T, 3) --> ['COUNT('], column(NL, DB, T), [')'].
selection_branch(NL, DB, T, 4) --> ['SUM('], column(NL, DB, T), [')'].
selection_branch(NL, DB, T, 5) --> ['AVG('], column(NL, DB, T), [')'].
selection_branch(NL, DB, T, 6) --> ['MIN('], column(NL, DB, T), [')'].
selection_branch(NL, DB, T, 7) --> ['MAX('], column(NL, DB, T), [')'].
nn_lm(selection_lm, [NL], Y, S_domain(Y), Prompt) :: selection(NL, DB, T) --> selection_branch(NL, DB, T, Y).
nn_lm(table_lm, [NL], T, table_domain(DB, T), Prompt) :: table(NL, DB, T) --> [].
nn_lm(column_lm, [NL], C, column_domain(DB, T, C), Prompt) :: column(NL, DB, T) --> token(C).
query(NL, DB) --> table(NL, DB, T), ['SELECT'], selection(NL, DB, T), ['FROM'], token(T).
```

A.2 Training and Evaluation

We train settings 1 and 2 end-to-end for 7 epochs using the Adam optimizer [10] with a batch size of 8 and a learning rate of $1e^{-3}$. For evaluation, we employ DeepStochLog inference with the $(\max, \times)$ semiring, i.e. the exact inference. Examples of *column_lm* and *selection_lm* inputs are listed in Table 7. The inputs of *table_lm* have the same format as those of *column_lm*.

Pre- and post-processing In pre-processing, we tokenize the ground truth SQL queries to sequences with list format. Table and column identifiers in the sequences are replaced by their semantic names [12]. The semantics names provided in Spider [34] are closer to natural expressions, which facilitate the understanding of language models. This replacement is also conducted for the facts in logic programs. After generation, we restore the identifiers in the output sequence to their original names and join the sequence to get the SQL query. This pre- and post-progressing are also performed in task 2 experiments.

B Task 2

B.1 Logic Program

Task 2 grammar covers two types of queries: 1) single selection and 2) two selections connected by the set operator "EXCEPT". The grammar for single-selection queries covers the "SELECT", "WHERE", "GROUP BY", and "ORDER BY" clauses including "DISTINCT" and aggregation functions in the "SELECT" clause, "HAVING" in the "GROUP BY" clause, and "ASC / DESC" and "LIMIT" in the "ORDER BY" clause. "WHERE" and "HAVING" allow one condition. For type 2), the two selection clauses have the format "SELECT [column] FROM [table]". The columns in the two selection clauses are currently restricted to foreign keys linking the two tables. Task 2 grammar is modeled as follows:

```
selection_domain(Y) :- member(Y, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]).
boolean_domain(Y) :- member(Y, [0, 1]).
where_operator_domain(Y) :- member(Y, ['=', '>', '<', '>=', '<=', 'LIKE']).
having_operator_domain(Y) :- member(Y, ['=', '>', '<', '>=', '<=']).
except_table_1_domain(DB, T) :- database_foreign_tables(DB, ForeignTables), member(T, ForeignTables).
except_table_2_domain(DB, T1, T) :- table_foreign_relations(DB, T1, ForeignTables), member(T, ForeignTables).
table_domain(DB, T) :- database(DB, Tables), member(T, Tables).
column_domain(DB, T, C) :- table(DB, T, Columns), member(C, Columns).
token(X) --> [X].
nn_lm(table_lm, [NL, State], T, table_domain(DB, T), Prompt) :: table(NL, DB, T, State) --> [].
nn_lm(column_lm, [NL, State], C, column_domain(DB, T, C), Prompt) :: column(NL, DB, T, State) --> token(C).
selection_branch(_, _, _, 0) --> ['*'].
selection_branch(_, _, _, 1) --> ['COUNT(*)'].
selection_branch(NL, DB, T, 2) --> column(NL, DB, T, 0).
selection_branch(NL, DB, T, 3) --> ['DISTINCT'], column(NL, DB, T, 0).
selection_branch(NL, DB, T, 4) --> ['COUNT('], column(NL, DB, T, 0), [')'].
selection_branch(NL, DB, T, 5) --> ['COUNT('], ['DISTINCT'], column(NL, DB, T, 0), [')'].
selection_branch(NL, DB, T, 6) --> ['SUM('], column(NL, DB, T, 0), [')'].
selection_branch(NL, DB, T, 7) --> ['AVG('], column(NL, DB, T, 0), [')'].
selection_branch(NL, DB, T, 8) --> ['MIN('], column(NL, DB, T, 0), [')'].
selection_branch(NL, DB, T, 9) --> ['MAX('], column(NL, DB, T, 0), [')'].
nn_lm(selection_lm, [NL], Y, selection_domain(Y), Prompt) :: selection(NL, DB, T) --> selection_branch(NL, DB, T, Y).
nn_lm(operator_lm, [NL, State], Y, where_operator_domain(Y), Prompt) :: where_operator(NL, State) --> token(Y).
where_branch(_, _, _, 0) --> [].
where_branch(NL, DB, T, 1) --> ['WHERE'], column(NL, DB, T, 1), where_operator(NL, 0), ['WHERE-VALUE'].
nn_lm(where_lm, [NL], Y, boolean_domain(Y), Prompt) :: where(NL, DB, T) --> where_branch(NL, DB, T, Y).
nn_lm(operator_lm, [NL, State], Y, having_operator_domain(Y), Prompt) :: having_operator(NL, State) --> token(Y).
having_branch(_, 0) --> [].
having_branch(NL, 1) --> ['HAVING'], ['COUNT(*)'], having_operator(NL, 1), ['HAVING-VALUE'].
nn_lm(having_lm, [NL], Y, boolean_domain(Y), Prompt) :: having(NL) --> having_branch(NL, Y).
groupby_branch(_, _, _, 0) --> [].
groupby_branch(NL, DB, T, 1) --> ['GROUP BY'], column(NL, DB, T, 2), having(NL).
nn_lm(groupby_lm, [NL], Y, boolean_domain(Y), Prompt) :: group_by(NL, DB, T) --> groupby_branch(NL, DB, T, Y).
limit_branch(0) --> [].
limit_branch(1) --> ['LIMIT'], ['LIMIT-VALUE'].
nn_lm(limit_lm, [NL], Y, boolean_domain(Y), Prompt) :: limit(NL) --> limit_branch(Y).
desc_branch(0) --> ['ASC'].
desc_branch(1) --> ['DESC'].
nn_lm(desc_lm, [NL], Y, boolean_domain(Y), Prompt) :: desc(NL) --> desc_branch(Y).
orderby_branch(_, _, _, 0) --> [].
orderby_branch(NL, DB, T, 1) --> ['ORDER BY'], column(NL, DB, T, 3), desc(NL), limit(NL).
nn_lm(orderby_lm, [NL], Y, boolean_domain(Y), Prompt) :: order_by(NL, DB, T) --> orderby_branch(NL, DB, T, Y).
query_type(NL, DB, 0) --> table(NL, DB, T, 0), ['SELECT'], selection(NL, DB, T), ['FROM'],token(T), where(NL, DB,
T), group_by(NL, DB, T), order_by(NL, DB, T).
nn_lm(table_lm, [NL, State], T, except_table_1_domain(DB, T), Prompt) :: except_table_1(NL, DB, T, State) --> [].
nn_lm(table_lm, [NL, State], T, except_table_2_domain(DB, T1, T), Prompt) :: except_table_2(NL, DB, T1, T, State) --> [].
query_type(NL, DB, 1) --> except_table_1(NL, DB, T1, 0), except_table_2(NL, DB, T1, T2, 1), {foreign_key(T1, T2, C1, C2)},
['SELECT'], token(C1), ['FROM'], token(T1), ['EXCEPT'], token(C2), ['FROM'], token(T2).
nn_lm(except_lm, [NL], Y, boolean_domain(Y), Prompt) :: query(NL, DB) --> query_type(NL, DB, Y).
```

B.2 Underlying Language Models

We employ 11 fine-tuned T5-small models. All models used in this work are from Huggingface [28].

table_lm and *column_lm* provide the probability distribution over the given table and column domain respectively. Their output domains vary with the database. The output domain of other language models is fixed. *except_lm*, *where_lm*, *groupby_lm*, *having_lm*, *order_lm*, *desc_lm*, *limit_lm* are used to produce the probability distribution on the existence of the corresponding SQL clause. *selection_lm* provides the probability distribution over 10 possible selection branches. *operator_lm* outputs the probability distribution over possible operators in "WHERE" and "HAVING" conditions. For the queries with two selection clauses connected by "EXCEPT", we call *table_lm* twice to get probability distributions over the possible substitutions for the table in each selection clause. The pair of columns in each selection clause are decided deterministically based on the pair of tables. Our current framework does not predict any value in SQL conditions. We assume the gold values are given. When we predict wrong conditions that cannot be mapped to the gold values, we assign the values to 1.

table_lm, *column_lm*, and *opertor_lm* can be used at different positions. *table_lm* can be called twice for the selection clause before "EXCEPT" and the one after "EXCEPT". *column_lm* is used for the column in the "SELECT",

"WHERE", "GROUP BY" and "ORDER BY" clauses. *operator_lm* is used for operators in "WHERE" and "HAVING" conditions. We add states in their inputs to help them distinguish different cases. Examples of *column_lm* inputs are shown in Table 7 to showcase the inputs with states. We also include examples of *where_lm* and *selection_lm* inputs in Table 7. *except_lm*, *where_lm*, *groupby_lm*, *having_lm*, *order_lm*, *desc_lm*, *limit_lm* shares a similar input format as *where_lm*.

B.3 Training and Evaluation

In the text-to-SQL task, the grammar can always be written unambiguously, which leads to only one possible derivation for the ground-truth query Q. With the negative log-likelihood loss function and all positive samples in dataset $\mathcal{D}$ (target probability t_i=1.0),

$$\begin{aligned}(1) &= \min_{p} \sum\nolimits_{(G_i\theta_i, Q_i)\in D} -log(\prod\nolimits_{r_j \in d(G_i\theta_i)=Q_i} p_j^{k_j}) \\ &= \sum\nolimits_{(G_i\theta_i, Q_i)\in D} \sum\nolimits_{r_j \in d(G_i\theta_i)=Q_i} \min_{p_j} -k_j log p_j\end{aligned}$$

As all the intermediate goals are observable given Q, the learning problem collapses to supervised training.

We fine-tune the T5-small models using the Adam optimizer. Table 6 shows the number of fine-tuning epochs, the batch size, and the learning rate for each model. The hyper-parameters are determined with cross-validation.

In evaluation, we perform the greedy inference to speed up the inference process. Instead of considering the re-normalized distributions obtained from the language models, the greedy inference takes the substitution with the largest probability.

Table 6. Fine-tuning hyper-parameters of our T5-small models.

	table	column	except	where	group by	having	order by	desc	limit	selection	operator
Epochs	5	16	3	14	4	2	7	10	3	7	17
Batch size	64	32	64	32	64	64	64	32	64	64	64
Learning rate	$1e^{-3}$	$5e^{-4}$	$5e^{-4}$	$5e^{-4}$	$1e^{-3}$	$5e^{-4}$	$1e^{-3}$	$5e^{-4}$	$5e^{-4}$	$1e^{-3}$	$1e^{-3}$

C Baselines

We fine-tune the vanilla T5-small baseline for 10 epochs using the Adam optimizer, a batch size of 32, and a learning rate $1e^{-3}$. T5-small + CFGs shares the same language models with ours T5-small + DCGs except the *column_lm*.

Without table unification, the domain of *column_lm* used in T5-small + CFGs covers all the columns in a given database. We fine-tune a T5-small model for *column_lm* in T5-small + CFGs using the Adam optimizer for 13 epochs with a batch size of 32, and a learning rate $5e^{-4}$. Table 8 shows the example inputs for T5-small and *column_lm* in T5-small + CFGs.

For the state-of-the-art models (DAIL-SQL [7], DIN-SQL [18], Graphix-T5 [13], and C3 [6]), we extract their predictions on the samples in our evaluation set from their official results for the full Spider development set [34].

Table 7. Examples of inputs of our T5-small models.

Task 1	column_lm	On average how large is the population of the counties? Answer 1 for county id, Answer 2 for county name, Answer 3 for population, Answer 4 for zip code, the answer should be Answer
	selection_lm	How many singers do we have? Answer 1 for *, Answer 2 for COUNT(*), Answer 3 for column, Answer 4 for COUNT(column), Answer 5 for SUM(column), Answer 6 for AVG(column), Answer 7 for MIN(column), Answer 8 for MAX(column), the answer should be Answer
Task 2	column_lm	What is the average hours across all projects? SELECT [column], Answer 1 for code, Answer 2 for name, Answer 3 for hours, the answer should be Answer
		Find the ids of all the order items whose product id is 11. WHERE [column], Answer 1 for order item id, Answer 2 for product id, Answer 3 for order id, Answer 4 for order item status, Answer 5 for order item details, the answer should be Answer
		Find the number of followers for each user. GROUP BY [column], Answer 1 for user id, Answer 2 for follower id, the answer should be Answer
		List all pilot names in ascending alphabetical order. ORDER BY [column], Answer 1 for pilot id, Answer 2 for name, Answer 3 for age, the answer should be Answer
	selection_lm	How many singers do we have? Answer 1 for *, Answer 2 for COUNT(*), Answer 3 for column, Answer 4 for DISTINCT column, Answer 5 for COUNT(column), Answer 6 for COUNT(DISTINCT column), Answer 7 for SUM(column), Answer 8 for AVG(column), Answer 9 for MIN(column), Answer 10 for MAX(column), the answer should be Answer
	where_lm	How many king beds are there? Answer 1 for empty, Answer 2 for WHERE, the answer should be Answer

Table 8. Examples of baseline inputs.

T5-small	Please show the categories of the music festivals with count more than 1. database is music_4. tables are artist, volume, music festival. columns in artist are artist id, artist, age, famous title, famous release date. columns in volume are volume id, volume issue, issue date, weeks on top, song, artist id. columns in music festival are id, music festival, date of ceremony, category, volume, result.
T5-small + CFGs (column_lm)	What is the average hours across all projects? SELECT [column], Answer 1 for ssn, Answer 2 for name, ..., Answer 7 for project, the answer should be Answer
	Find the ids of all the order items whose product id is 11. WHERE [column], Answer 1 for customer id, Answer 2 for customer name, ..., Answer 27 for order item id, the answer should be Answer
	Find the number of followers for each user. GROUP BY [column], Answer 1 for user id, Answer 2 for follower id, ..., Answer 11 for followers, the answer should be Answer
	List all pilot names in ascending alphabetical order. ORDER BY [column], Answer 1 for pilot id, Answer 2 for name, ..., Answer 28 for aircraft id, the answer should be Answer