Reliable Answers for Recurring Questions: Boosting Text-to-SQL Accuracy with Template Constrained Decoding

SMIT JIVANI, SARVAM MAHESHWARI, and SUNITA SARAWAGI, Department of Computer Science and Engineering, Indian Institute of Technology Bombay, India

Large language models (LLMs) have revolutionised Text-to-SQL generation, allowing users to query structured data using natural language with growing ease. Yet, real-world deployment remains challenging, especially in complex or unseen schemas, due to inconsistent accuracy and the risk of generating invalid SQL.

We introduce Template Constrained Decoding (TeCoD), a system that addresses these limitations by harnessing the recurrence of query patterns in labeled workloads. TeCoD converts historical NL-SQL pairs into reusable templates and introduces a robust template selection module that uses a fine-tuned natural language inference model to match or reject queries efficiently. Once the template is selected, TeCoD enforces it during SQL generation through grammar-constrained decoding, implemented via a novel partitioned strategy that ensures both syntactic validity and efficiency. Together, these components yield up to 36% higher execution accuracy than in-context learning (ICL) and 2.2× lower latency on matched queries.

CCS Concepts: • **Human-centered computing** → **Natural language interfaces**; • **Information systems** → **Structured Query Language**; **Question answering**.

Additional Key Words and Phrases: Text-to-SQL; Constrained Generation; Partitioned Decoding; Structured Code Generation

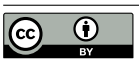
ACM Reference Format:
Smit Jivani, Sarvam Maheshwari, and Sunita Sarawagi. 2025. Reliable Answers for Recurring Questions: Boosting Text-to-SQL Accuracy with Template Constrained Decoding. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 357 (December 2025), 26 pages. https://doi.org/10.1145/3769822

1 Introduction

Accessing databases via natural language queries (NLQs) has been a long-standing goal of the database community [16, 20, 25]. Recently, LLMs with their superior capability of natural language understanding and code generation, have achieved great strides in the accuracy of Text-to-SQL generation as seen via public benchmarks [19, 35]. The benchmark numbers are averaged over multiple database schemas, and typically evaluated zero-shot on unseen schema. However, database-specific accuracy shows significant variation with changing schema. In Figure 1 we show accuracy on 11 schemas. For some databases, the accuracy is low enough to be of little use in practical systems. Such an experience is common for enterprises, whose databases are private to frontier LLMs. Consequently, the enterprise may be willing to organize workload of NLQs with expert provided correct SQL, and adapt the LLM to increase its Text-to-SQL accuracy to acceptable levels.

Currently, there are two options for adaptation: fine-tuning and in-context learning. Fine-tuning requires a lot of labeled data up-front, is unaffordable for small enterprises, and leads to forgetting of other tasks. These limitations have caused great interest in non-fine-tuning based adaptation

Authors' Contact Information: Smit Jivani, smitjivani@iitb.ac.in; Sarvam Maheshwari, sarvam@iitb.ac.in; Sunita Sarawagi, sunita@iitb.ac.in, Department of Computer Science and Engineering, Indian Institute of Technology Bombay, Mumbai, India.



ACM 2836-6573/2025/12-ART357
https://doi.org/10.1145/3769822

methods. In-context learning (ICL) is currently the go-to method for adapting an LLM to new databases without fine-tuning [23, 29]. ICL just requires modifying the input prompt with a few examples of labeled related Text-SQL pairs to adapt the model on-the-fly for each user query. Often, ICL leads to a significant boost in accuracy, as we show in Figure 1.

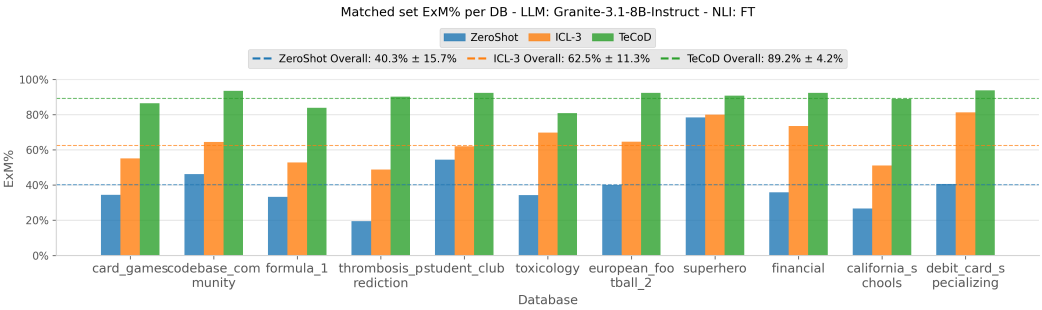


Fig. 1. Execution Match Accuracy(%) per database for recurring questions (questions with a matching template). Observe that TeCoD provides close to 90% accuracy across almost all databases, whereas both ZeroShot and ICL are much worse and exhibit high variance.

In this paper we propose methods that goes beyond in-context learning in harnessing labelled data of an enterprise. Our work is based on two premises: (1) The workload of an enterprise often contains clusters of highly similar SQLs. We observed this pattern in the real workload of a very large bank, discussed more in Section 5.1. The user's question (NLQ) could be different because of the inherent diversity of natural language, but the generated SQL is often highly similar to previously seen SQLs. (2) In-context learning disappoints in how well it harnesses the related queries. We show two examples in Table 1, where the LLM fails to generate the correct SQL even in the presence of an in-context example with SQL differing only by a constant.

We design a system TeCoD, to go beyond in-context learning to harness closely related queries to improve accuracy of SQL generation. Our core idea is to convert previously seen labeled Text-SQL pairs to templatized Text-SQL forms to foster greater match to future queries. When a future query matches one of the stored templates, TeCoD generates the SQL via a dedicated template constrained decoding. We show that accuracy of matched queries jumps from approximately 60% with ICL to almost 90% via our constrained decoding. This implies that recurring queries can be executed with significantly higher reliability than baseline ICL methods. In addition, we take advantage of the templatized form to achieve almost a factor of two improvements in inference throughput.

We encountered two primary challenges in the implementation of TeCoD. The first challenge was designing the matcher module for accurately matching queries to a stored pool of templatized Text-SQL pairs. We need to reject queries that do not match any template in the pool and choose the correct template from the pool for the rest. Since a wrongly matched template is guaranteed to provide the wrong SQL with constrained decoding, high accuracy in this step is crucial. We propose three strategies to boost the accuracy of the matcher beyond simple thresholded cosine similarity of sentence embeddings: (1) casting template match as a natural language inference problem, (2) masking parts of the user question, (3) generating multiple synthetic paraphrases of the original labeled Text-SQL pair to provide multiple text annotation to a shared template. Together, these strategies obtained a jump in selection/rejection accuracy from 73% to 91% beyond the baseline.

The second challenge was efficiently and accurately enforcing the template constraints as the LLM generates the SQL for a user query. Recently, many libraries have been developed for constraining

Example 1:
Question: How many K-12 schools in Contra Costa register more than 420 free meals but have free or reduced-priced meals numbering under 610?

```
SELECT COUNT(CDSCode) FROM frpm WHERE `County Name` = 'Contra Costa' AND `Free Meal
     Count (K-12)` > 420 AND `FRPM Count (K-12)`< 610
```

———————

Question: In Los Angeles how many schools have more than 500 free meals but less than 700 free or reduced price meals for K-12?

```
SELECT COUNT(CDSCode) FROM frpm WHERE `County Name` = 'Los Angeles' AND `Free Meal
    Count (K-12)` > 500 AND `Enrollment (K-12)` < 700
```

Example 2:
Question: What is the total number of home team goals scored by Eric Djemba-Djemba?

```
SELECT SUM(t2.home_team_goal) FROM Player AS t1 INNER JOIN match AS t2 ON t1.
    player_api_id = t2.away_player_9 WHERE t1.player_name = 'Eric Djemba-Djemba'
```

———————

Question: Aaron Lennon refers to player_name = 'Aaron Lennon'; How many home team goal have been scored by Aaron Lennon?

```
SELECT SUM(t2.home_team_goal) FROM Player AS t1 INNER JOIN match AS t2 ON t1.
    player_api_id = t2.home_player_11 WHERE t1.player_name = 'Aaron Lennon';
```

Table 1. Examples where the ICL method generated wrong output in the presence of a very similar example. In the examples, the first NLQ-SQL pair is the ICL example, followed by the user NLQ and predicted SQL.

text generated by LLMs to satisfy constraints specified as a regular expression or context-free grammar [3, 10, 32]. We show how to cast the template as a flexible grammar that provides better agreement with the LLMs formatting biases by converting the masked SQL into a regular expression derived from the SQL grammar. We present an efficient two-phase decoding algorithm for efficient constrained SQL generation. In the first phase, we incur a one-time overhead to pre-compile the template grammar to generate LLM-aligned token sequence corresponding to the query-invariant part of the SQL template. In the second phase, we efficiently fill in only the query-specific masked literals. Overall, these lead to similar accuracy with decoding time reduced to $0.4 - 0.6\times$ of the library default.

Contributions. (1) Introducing the paradigm of template constrained decoding for recurring queries to address the accuracy and latency challenges of Text-to-SQL generation in an enterprise. (2) Design of an accurate template matcher module that decides if natural language question corresponds to an SQL that conforms to one of the stored templates. (3) Accurate and efficient adaptation of grammar constrained decoding libraries for template constrained SQL generation. (4) Evaluation on two SOTA Text-SQL benchmarks and five LLMs yielding an accuracy jump from an average of 60% (with ICL) to almost 90% (with TeCoD) on a workload of recurring queries in the BIRD databases.

2 Related Work

Text-to-SQL generation has been a very active and fast-progressing research area, with significant progress made in various areas, including design of models [18, 30], schema and value subsetting [13, 17], prompts and inference pipelines like CoT, consensus based reranking [9, 15, 23, 24, 38],

and other reasoning-based methods [37]. These techniques contribute to improving the baseline performance of Text-SQL systems across schema. For enterprises, where high accuracy is of paramount importance, and where memory of users and deployments are easily available, workload-drive customization of generic designs is of great interest. We review existing methods of adapting pre-trained LLMs with schema-specific workloads.

Workload-driven Customization. One class of methods proposes to fine-tune pre-trained LLMs to the Text-SQL tasks. Examples include CodeS [18] that focused on fine-tuning for cross-schema generalization. One challenge with fine-tuning models for a specific target database is collecting labeled data up front. Some methods propose to augment with synthetic examples [1, 31]. However, with the advent of LLMs, the focus shifted to on-the-fly customization without model fine-tuning. LLMs naturally support In-Context Learning [5] where examples of Text-SQL pairs provided within the context have been found to adapt the LLM to SQLs of a target database [18, 33, 38]. Other methods of adaptation include case base reasoning, and these have been found to be effective for generating SQL as a relational algebra tree [28].

Template-based generation Templates have been harnessed for reducing the complexity of SQL generation in many prior systems. ZeroNL2SQL [7] uses a small language model to create candidate templates, and then uses a larger LLM to generate the final SQL via soft prompting and iterative feedback. CatSQL [8] uses a custom deep learning model to generate templates and fill the slots in the template, followed by semantic correction and post-processing of the generated SQL to fix the errors. AmbiQT [4] harnesses templates for generating structurally diverse SQLs. However, we are not aware of any prior work that proposes to harness existing labeled data as templates for more accurate SQL generation for similar queries.

Constrained-decoding/grammar-guidance Grammar- constrained decoding techniques ensure syntactically valid generation while leveraging the semantic understanding capabilities of large language models (LLMs). For SQL generation PICARD [27] pioneered this paradigm by integrating incremental parsing constraints during autoregressive decoding, dynamically rejecting tokens that violate SQL grammar rules and significantly improving execution validity. Recently, more general forms of constrained generation is supported by libraries such Transformer-CFG [10], DOMINO [3] and Outlines [32]. These differ in how they balance validity guarantees with computational efficiency through hybrid static/dynamic analysis of grammatical structures. Transformer-CFG [10] introduced explicit modeling of context-free grammar (CFG) states through finite state machine (FSM) representations. This approach parses the entire vocabulary against the current FSM state, masks invalid token IDs, and updates the state based on sampled tokens - ensuring syntactic correctness at the cost of substantial inference overhead due to real-time FSM transitions and vocabulary-wide validity checks. Recent advancements like DOMINO [3] address this efficiency challenge through offline precomputation of prefix trees (tries) for each FSM state. By encoding valid token sequences in trie structures during preprocessing, DOMINO improves the latency of validity checks during decoding using tries, reducing computational complexity while maintaining grammatical constraints. Outlines [32] is another open-source Python library. We are using Outlines as part of our system to generate SQL, but we innovate in how we efficiently invoke the library for template constraining.

3 Our approach

Problem Statement. We are given a database DB with schema S on which we wish to support natural language querying. Let M denote an instruction-tuned LLM that given any user's natural language question (NLQ) $\mathbf{q}$ and the schema S, can generate an SQL $\hat{\mathbf{y}}$ for $\mathbf{q}$. The default LLM may not provide high accuracy of conversion of NLQs to SQL. We assume that there exists a workload

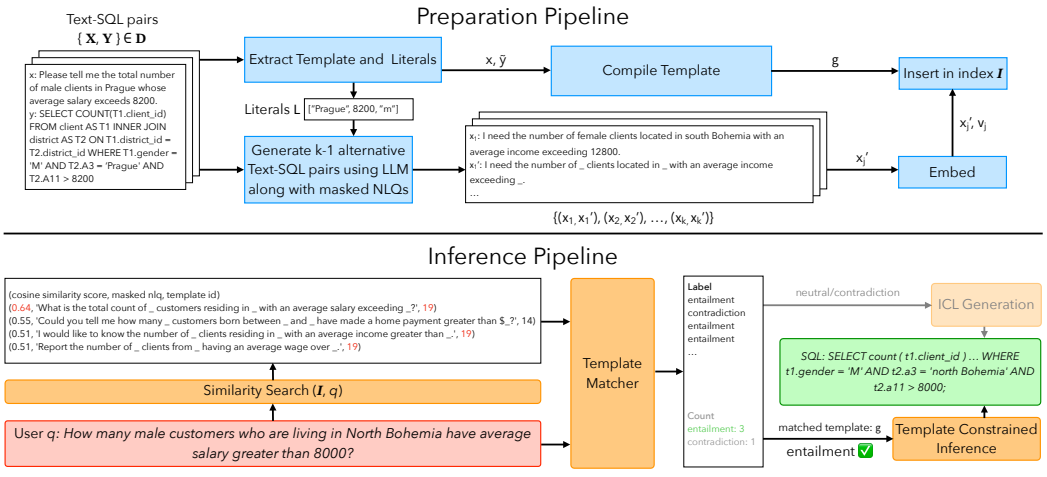


Fig. 2. System Architecture of TeCoD. The top part shows the processing done on labeled queries to extract and index templates. The bottom part shows steps during inference of each user query. See Figure 3 for Template Compilation and Template Constrained Inference.

of previous NLQs along with correct expert provided SQLs $D = \{(\mathbf{x}^1, \mathbf{y}^1), \ldots (\mathbf{x}^N, \mathbf{y}^N)\}$. N may not be large enough to perform supervised fine-tuning of M to the schema and queries of this DB. Our goal is to harness alternative strategies of customization.

A baseline method: In-Context Learning. A baseline method is to retrieve labeled Text-SQL pairs from D based on similarity of $\mathbf{q}$ with corresponding text $\mathbf{x}^i$ and include them as in-context examples in the prompt. As we show in Figure 1, including only a few related examples in the prompt can significantly enhance accuracy of the generated SQL. However, while overall accuracy improves we also observe several cases where even when a highly related query is present in the context, the LLM fails to generate the correct SQL. Two examples appear in Table 1

Our proposed method TeCoD is designed to more aggressively harness related examples in available labelled workload D.

Overview of TeCoD. We present an overview of TeCoD in Figure 2. First, in the preparation phase we convert each (NLQ $\mathbf{x}$,SQL $\mathbf{y}$) pair in D to a templatized form to allow better match. These are inserted in an index $\mathcal{I}$ for efficient retrieval in response to user queries. In Section 3.1 we provide details of this step. During inference, given a user's natural language question (NLQ) $\mathbf{q}$, TeCoD first invokes a template matcher to decide if $\mathcal{I}$ contains a template that conforms to the (unknown) correct SQL of q. If a valid template is found, TeCoD generates the SQL constraining it to follow the matched template. Otherwise, the SQL is generated using the standard method using in-context examples selected from D or zero-shot depending on the baseline performance of the model. The template selection and matching module is a critical component of this pipeline, and we describe its design in Section 3.2. Another interesting component is how to decode by constraining as per the chosen template. We describe the design of the template constrained decoding in Section 3.3.

3.1 Template Extraction and Indexing

We process each NLQ $\mathbf{x}$, SQL $\mathbf{y}$ pair from D into a templatized form in two steps. First, we convert the SQL $\mathbf{y}$ into a template $\tilde{\mathbf{y}}$ and compile for efficient enforcement, and second we generate natural

Table 2. Example illustrating the templatization of Text-SQL Pairs after augmentation with synthetic NLQs

Original Text-SQL	Templatized and augmented
NLQ **x**: How much, in total, did client number 617 pay for all of the transactions in 1998?	Masked NLQ 1: What was the total amount paid by client number [number] for all transactions in [string]? Masked NLQ 2: How much did client number [number] pay altogether for every transaction in [string]?
SQL **y**: SELECT SUM(T3.amount) FROM client AS T1 INNER JOIN disp AS T4 ON T1.client_id = T4.client_id INNER JOIN account AS T2 ON T4.account_id = T2.account_id INNER JOIN trans AS T3 ON T2.account_id = T3.account_id WHERE STRFTIME('%Y', T3.date)= '1998' AND T1.client_id = 617	Template **y**: select sum(t3.amount) from client as t1 inner join disp as t4 on t1.client_id = t4.client_id inner join account as t2 on t4.account_id = t2.account_id inner join trans as t3 on t2.account_id = t3.account_id where strftime([string], t3.date) = [string] and t1.client_id = [number]
NLQ **x**: Who among KAM's customers consumed the most? How much did it consume?	Masked NLQ 1: Which customer of [string] had the greatest level of consumption? What quantity did they consume? Masked NLQ 2: In terms of consumption, who stands out among [string]'s customers? How much did they consume?
SQL **y**: SELECT T2.CustomerID, SUM(T2.Consumption) FROM customers AS T1 INNER JOIN yearmonth AS T2 ON T1.CustomerID = T2.CustomerID WHERE T1.Segment='KAM' GROUP BY T2.CustomerID ORDER BY SUM(T2.Consumption) DESC LIMIT 1	Template **y**: select t2.customerid, sum(t2.consumption) from customers as t1 inner join yearmonth as t2 on t1.customerid = t2.customerid where t1.segment = [string] group by t2.customerid order by sum(t2.consumption) desc limit [number]

language annotations to the template $\tilde{\mathbf{y}}$ so that these annotations can serve as search keys for matching with future natural language queries. We describe each of these steps next:

Template Extraction and Compilation. First, we convert the SQL $\mathbf{y}$ into a template $\tilde{\mathbf{y}}$ that is more likely to be shared by future queries. In this paper, we restrict the template to be the SQL with just constants and literals masked. Thus, a template in our definition is a parameterized SQL query. Two example SQLs and their corresponding templatized forms appear in Table 2.

Let L denote the set of literals present in SQL $\mathbf{y}$. Extracting such literals from SQL is easy using an off-the-shelf parser like SQLGlot [22]. Next, we compile the templatized SQL into a flexible grammar $\mathbf{g}$ to be used during constrained decoding. More details of this step appear in Section 3.3.

Natural Language Annotations for Template. To provide a covering set of annotations, we generate with the help of the LLM, synthetic Text-SQL pairs $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_K, \mathbf{y}_K)$ such that each SQL $\mathbf{y}_i$ follows the template $\tilde{\mathbf{y}}$. This implies that each $\mathbf{y}_i$ differs from the original SQL $\mathbf{y}$ only in values of literals. Let $A(\tilde{\mathbf{y}})$ denotes the generated Text-SQL pairs including the original $(\mathbf{x}, \mathbf{y})$. We convert $\mathbf{x}_i \in A(\tilde{\mathbf{y}})$ into a natural language annotation for $\tilde{\mathbf{y}}$ by masking away tokens in $\mathbf{x}_i$ that refer to constants literals as follows. Let L be the literals in $\mathbf{y}_i - \tilde{\mathbf{y}}$ that we extract using an SQL parsing library. We mask the mentions of literals L in the question to increase its match with future queries with differing literals. Masking literal mentions in natural language is challenging. We use the following approach: We first sort the literals in L by length in descending order so that longer literals

are matched first. To mask out in $\mathbf{x}_i$ the mention of a literal $\ell \in L$, we first look for case-insensitive exact matches of ℓ in $\mathbf{x}_i$. After that, we do fuzzy matching using the RapidFuzz [2] Python package for approximate matching. We denote the masked NLQ as $\tilde{\mathbf{x}}_i$. At the end of this process the masked $\tilde{\mathbf{x}}_i$ should be relevant to $\tilde{\mathbf{y}}$. Table 2 shows examples of two Text-SQL pairs and their corresponding masked templatized forms.

Finally, we create vector embedding of all templated NLQ $\tilde{\mathbf{x}}$s using a neural sentence embedding model like NV-Embed-v2 [14]. The embedding is used to create the key to a hash-index with value as the template-id which in turn leads us to the corresponding compiled grammar. We use $\mathcal{I}$ to denote the index of masked NLQ and template-id pairs.

3.2 Template Selection

Given a new user NLQ $\mathbf{q}$, we need to find from $\mathcal{I}$ a template $\tilde{\mathbf{y}}$, if any, that would fit the correct SQL of $\mathbf{q}$. If no matching template is found, we default to the standard path of generating SQL using soft hints in the form of in-context examples selected from D. If a matching template $\tilde{\mathbf{y}}$ is found, we use the compiled grammar $\mathbf{g}$ to generate the SQL using template constrained decoding as described in Section 3.3. Since this step forces the generation of the SQL to follow the prescribed template, it is important to perform the template matching and selection step with high accuracy. The core module in this step is a template matcher model that we describe next.

Algorithm 1 Template Search with NLI Validation and Selection

Input: Natural Language Question (q), NLQ-Template Index $\mathcal{I}$, Top-k results (k)
Output: Template Matched (`matched`), Template ID (t_id)
`top_k` $\leftarrow$ `sort(similarity_search(`$\mathcal{I}, x, k$`), by=cosine similarity, desc)`
`best_match` $\leftarrow$ `top_k[0]`
`t_id` $\leftarrow \mathcal{I}$`[best_match]`
`nli_results` $\leftarrow$ [`NLI(`$x, \tilde{\mathbf{x}}$`)` **for** $\tilde{\mathbf{x}}$ `in top_k` **if** $\mathcal{I}[\tilde{\mathbf{x}}]$ `== t_id`]
`nli_label` $\leftarrow$ `majority_vote(nli_results)`
`matched` $\leftarrow$ (`nli_label == 'entailment'`)
Return `matched, t_id`

Template Matcher. The template matcher needs to decide if a user NLQ $\mathbf{q}$ matches a stored template text $\tilde{\mathbf{x}}$ such that the (unknown) SQL of $\mathbf{q}$ would follow the template $\tilde{\mathbf{y}}$. We found that pre-trained LLMs were not accurate for such reasoning. Also, just measuring the similarity of $\mathbf{q}$ and $\tilde{\mathbf{x}}$ using popular methods like cosine similarity of their respective sentence embeddings was not accurate enough. The task of deciding whether a template fits the correct SQL for an input NLQ requires more nuanced modeling. Towards this end we trained a dedicated template matching model by repurposing a natural language inference (NLI) model. An NLI model takes as input a pair of natural language sentences s_1, s_2 and decides if the logic in sentence s_1 entails, contradicts, or is neutral with the logic in sentence s_2. Unlike embedding based models, state-of-the-art NLI models capture fine-grained interaction between the words of the two sentences using a Transformer with bidirectional attention to decide on these labels. In our application, one of the sentences $\tilde{\mathbf{x}}$ is a NLQ with holes corresponding to the masked literals, and the other is the user NLQ $\mathbf{q}$. For example, in Figure 2 we need to match the user NLQ $\mathbf{q}$ correctly to the first, third and fourth masked annotation, and not the second. We fine-tuned a pre-trained NLI model to this form of the input. Since we already generated alternative NLQs for a template, we create pairs out of these and call these positive pairs. Next, for each NLQ, we found the closest NLQs from a different template, and call these as negative NLQs. These examples across all templates are used to fine-tune existing

pre-training NLI models like BERT[6]. We further increase robustness of the matcher to literal masking errors by training with both masked and un-masked NLQs. We present further details of the training process, and the accuracy gains with a trained matcher model in the experiment section.

Further, to avoid invoking the template matcher model on every stored NLQ in $\mathcal{I}$, we first filter Top-K matched NLQs using cosine similarity. The template-id with the closest masked NLQ based on cosine is filtered. The NLI model is invoked on each masked NLQ of the template. We get the NLI-label (entailment, neutral, contradiction) along with its probability for each pair. The final NLI label for a template is the majority of the predicted labels. The overall pseudocode occurs in Algorithm 1.

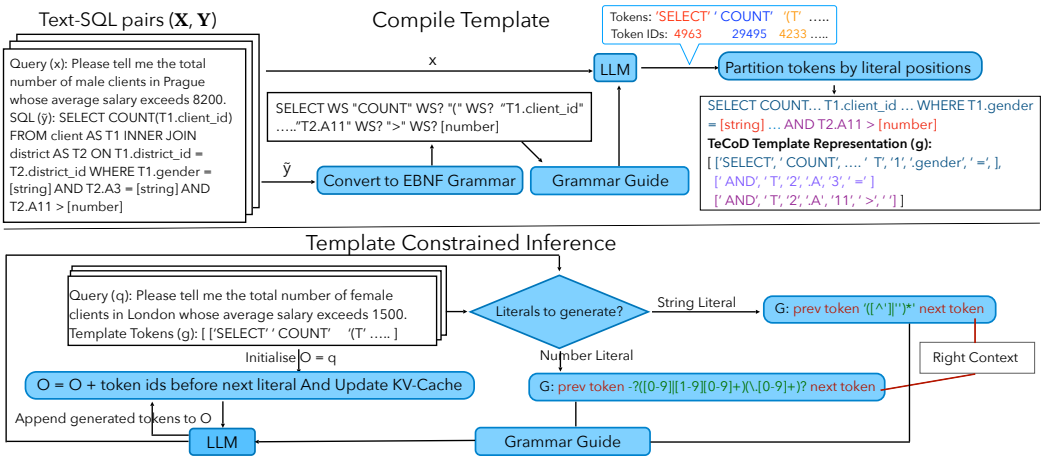


Fig. 3. Template Compilation and Template Constrained Inference. Top part shows the one-time process of converting any template $\tilde{\mathbf{y}}$ to a compiled representation g. Bottom part shows the iterative process to generate the template constrained SQL given user query $\mathbf{q}$ and a selected template's grammar g.

3.3 SQL Generation

Based on the NLI label, we decide whether to do template constrained decoding or unconstrained generation. In both cases, we include in-context-learning (ICL) examples in the form of demonstrations of pairs of user question and corresponding SQL. We next describe how we perform template constrained decoding.

We are given a selected template $\tilde{\mathbf{y}}$ and a user NLQ $\mathbf{q}$. Our goal is to generate the SQL for $\mathbf{q}$ while adhering to the template $\tilde{\mathbf{y}}$. Providing an LLM with template $\tilde{\mathbf{y}}$ along with instructions to follow the template does not guarantee that LLM will always adhere to that template as we will show in the experiment section. Consequently, we modify LLM's decoding mechanism to be guided by the grammar-constrained decoding (GCD) rules. We present a brief background of how recent GCD methods work and then describe our adaptation to the template constrained generation task.

Background: Grammar constrained decoding (GCD) in LLMs. Given an LLM M and grammar G which can be expressed as a regular expression or a context free grammar, recent libraries like Transformer_CFG [10] and Outlines [32] constrain LLM generated text to be valid as per the grammar G. The LLM generates the text token-by-token using auto-regressive decoding. At the t-th step of generation, let $y_1, \ldots, y_{t-1}$ denote the tokens generated so far, and let $P(y|y_1, \ldots, y_{t-1})$

denote the LLM's output probability distribution over its token vocabulary. Normally, the LLM would sample a token with high probability from this distribution to get the next token, y_t. Instead with GCD, the grammar G is consulted to mask from $P(y_t|y_1, \ldots, y_{t-1})$ any token that is invalid as per the grammar G. Generation stops when y_t is a special end-of-sequence token. These libraries mask invalid token ids and maintain the state of allowed outputs efficiently while generating a token at each timestep.

A template $\tilde{\mathbf{y}}$ in our case is an SQL with zero or more empty slots to be filled with string or number literals. Here is a baseline method for harnessing the LLM to infer the slot values to generate the SQL for the user query $\mathbf{q}$. Express the template as a regular expression such as shown in Table 3 (Row #3), and simply invoke existing GCD libraries to generate the SQL. A pseudocode is given in Algorithm 2 as a reference. The statements in blue show changes to the default LLM decoding algorithm to handle template constraints.

Algorithm 2 Template Constrained SQL Generation

Input: LLM M, Grammar constraining library C, SQL Template $\tilde{\mathbf{y}}$, Schema S, User query $\mathbf{q}$
$\mathbf{g} \leftarrow$ Express_As_Grammar($\tilde{\mathbf{y}}$)
Guide $\leftarrow$ Initialize state of C with $\mathbf{g}$
O_1 = Start of sequence token.
for t=1 to Max-SQL-length **do**
  $p(y) \leftarrow$ Next token distribution from LLM $M(S, \mathbf{q}, O_1 \ldots, O_t)$
  $m \leftarrow$ Get mask of allowed tokens from $C(\mathit{Guide}, O_1 \ldots, O_t)$
  $O_{t+1} \leftarrow$ sample next token from $p(y) \circ m$
  If O_{t+1} is EOS, Exit loop.
end for
return Decode $O_1 \ldots, O_t$ to SQL string.

There are two problems with this approach: style mismatch and wasteful LLM invocations that we elaborate on next.

Style Mismatch. The specific formatting of the SQL used in the template may not be compatible with the SQL formats the LLM may have seen in its training corpus. We found that each LLM has its own formatting and SQL styling preference such as case of keywords, use of aliases, punctuation, and white spaces between keywords. In Table 4 we show multiple style in which two different LLMs generate the same SQL. If we generate a fixed grammar that restricts the SQL generated to follow the string format in a given template, the LLM may not be accurate in generating the correct completions for the empty slots. We address this limitation by converting the template into a more elaborate SQL grammar that captures all surface forms of the different but syntactically equivalent ways in which the same SQL can be represented. An example of such a grammar is shown in Table 3 under the name of Flexible Template. Here we express the SQL template $\tilde{\mathbf{y}}$ as a regular expression where SQL keywords, operators etc are replaced with corresponding non-terminals. Every SQL keyword is allowed to be expressed in many different cases, and the whitespace between two keywords is flexible. Although tools like `sqlglot.qualify` can be used to normalize SQL queries by enforcing consistent aliasing, keyword casing, and formatting, we found that such normalization may not align with the LLM's SQL style. Such misalignment often leads to reduced accuracy when the LLM is used to fill the template. We show that with the flexibility induced by the SQL grammar, the generated SQL accommodates the different surface forms (as shown in Table 4) in which the initial template is expressed. However, this flexibility comes with run-time overheads since for

```
ỹ: SELECT * FROM Office WHERE Name = [String] Limit [Number];
Grammar:
STRING_RULE: '([^'|'']*)'
SINGLEDIGIT: [0-9]
NUMBER_RULE: ("-"? (SINGLEDIGIT | [1-9] SINGLEDIGIT*)) ("." SINGLEDIGIT+)? ([eE] [+-]?
     SINGLEDIGIT+)?
WS: [ \t\n\r]
SELECT_RULE: "select" | "SELECT" | "Select"
FROM_RULE: "from" | "FROM" | "From"
WHERE_RULE: "where" | "WHERE" | "Where"
LIMIT_RULE: "limit" | "LIMIT" | "Limit" ....

Fixed Template:
start: "SELECT" " " "*" " " "FROM" " " "Office" " " "WHERE" " " "Name" " " "=" " " [
    STRING_RULE] " " "Limit" " " [NUMBER_RULE] ";"

Flexible Template:
start: WS? SELECT_RULE WS "*" WS FROM_RULE WS "Office" WS WHERE_RULE "Name" WS "=" WS
    STRING_RULE LIMIT_RULE NUMBER_RULE (WS? | ";"?)
```

Table 3. Two different grammars for a template ỹ: (1) Fixed and (2) Flexible. Quoted strings (e.g."Name", " ") are fixed text to be emitted directly in the SQL output, while elements like SELECT_RULE are grammar rules expanded during decoding.

```
Llama: SELECT song_name FROM singer WHERE AVG > ( SELECT AVG(age) FROM singer );

Granite: SELECT song_name\nFROM singer\nWHERE age > (SELECT AVG(age) FROM singer);

CodeS: SELECT song_name FROM singer WHERE age >(SELECT avg(age) FROM singer)
```

Table 4. This example shows differences in LLM formatting preferences such use of \n or space, small or capital case for SQL function AVG, optional ';', spacing around brackets.

every generated SQL token the LLM needs to express its preference via its token distribution as shown in Algorithm 2.

Reducing LLM invocation cost via Partitioned Decoding. In a template, typically a large portion stays the same across user queries, only the slots are potentially query dependent. We address this limitation by designing a partitioned generation method that proceeds in two phases. First, in a one-time template compilation phase, we partition the template grammar ỹ into the parts that are independent of user query **q** and masked literals that are query-specific. We invoke the baseline whole grammar GCD method (Algorithm 2) once with the full grammar $G(\tilde{y})$ and remember the token-id sequences for the static parts. Second, at inference time, for each user query **q** conditioned on the static token-ids, the LLM generates the literals using GCD with only the string or number literal as specified. One subtle challenge with such partitioned generation arises from how the LLM tokenizes strings. In a template like "select * from office where (name = [String]) Limit [Number];" it is clear that only the literals within the box bracket need query-specific constrained generation. However, the LLM's preferred tokenization may straddle across the two partitions. For

example the LLM may have created a single token "1;" but partitioned generation will not allow generation of tokens that straddle partition boundaries. Such an issue appears in both the right and left context. To handle this problem we apply a simple fix: we move boundary tokens from the static partitions as left and right context around the literals that are GCD generated. We will show in Section 5.5 that such contextualization during literal generation is crucial for accurate SQL generation.

The overall pseudocode appears in Algorithm 3. To avoid repeated full-template decoding, we first precompute token IDs for the template using Algorithm 2 which is a one time step and then partition them into static segments and literal slots g (TeCoD Template Representation) as shown in figure 3. At inference, given a user query $\mathbf{q}$, only the slot values are generated via GCD. For each literal slot, we initialize with left and right context tokens (extracted respectively in lines 11 and 12). We then apply constrained decoding using regular expression for string or number literals as needed. The KV-cache from prior static tokens is reused for efficiency by avoiding repeat computation of key-value vectors. Table 7 provides the latency comparisons between standard constrained decoding and our two-phase approach. We discuss computational overhead of GCD in Section 5.5 where we compare latency overhead of two phase decoding with standard GCD and unconstrained generation.

Our approach builds on the strengths of CFG-based methods while optimizing for their computational inefficiencies for template-constrained generation. By narrowing the focus of the LLM to only the masked portions of the query and leveraging regex-guided decoding, we achieve a balance between efficiency, accuracy and practicality, which makes it particularly suitable for real-world applications with strict latency requirements.

4 Experiment Setting

We evaluate the performance of TeCoD in terms of both execution accuracy of the generated SQL and running time. We consider two kinds of test queries: `Matched` where a matching template is present in $\mathcal{T}$, and `Unmatched` where no matching query is present. Using these, our goal is to answer the following research questions using this empirical evaluation.

Research questions.

(1) RQ0: Is there sufficient evidence of template reuse in real-life query workloads?
(2) RQ1: For queries in the `Matched` set does TeCoD provide any gains beyond existing option of adapting with in-context learning.
(3) RQ2: What is accuracy of template selection for `Matched` and `Unmatched` queries?
(4) RQ3: What is the impact of various aspects of our template matching module — masking literals, augmentation with synthetic NLQs, use of a fine-tuned NLI model.
(5) RQ4: What is the accuracy boost with a flexible template enforcement beyond a string based template constraint?
(6) RQ5: What is the running time overhead of TeCoD?

4.1 Dataset

We present our evaluation on two popular Text-to-SQL benchmarks: BIRD-SQL [19] and Spider [36]. In all cases we evaluate on the Dev set of the benchmark. Unfortunately, these benchmarks are curated without any regard to preserve the actual frequency of occurrence of repeated queries in the workload. We handle this limitation by experimenting on two kinds of datasets as described below:

Algorithm 3 Efficient Partitioned Constrained Decoding

Input: LLM M, Grammar constraining library C, SQL Template $\tilde{\mathbf{y}}$, Schema S, User query $\mathbf{q}$
Generating LLM Aligned Token ID Sequence (Offline):
$O_1 \ldots, O_t \leftarrow$ Invoke Algorithm 2$(M, C, \tilde{\mathbf{y}}, S, \mathbf{q})$
$g \leftarrow$ Partition By Literals$(O_1 \ldots, O_t)$
//TeCoD Template Representation
Inference:
$num\_regex \leftarrow$ `-?([0-9]|[1-9][0-9]+)(\.[0-9]+)?`
$str\_regex \leftarrow$ `'([^']|'')*'`
$O \leftarrow$ Tokens for $S + \mathbf{q}$
$cache \leftarrow \phi$ //kv_cache
for $i = 0$ to $\#literals - 1$ **do**
  $next\_token \leftarrow$ Pop first token from g_{i+1}
  $prev\_token \leftarrow$ Pop last token from g_i
  if $literals_i$ is string **then**
    $guide_{str} \leftarrow$ Initialize C with $(prev\_token + str\_regex + next\_token)$
    $O \leftarrow$ LLM $M(O + g_i, guide_{str}, cache)$
  else if $literals_i$ is number **then**
    $guide_{num} \leftarrow$ Initialize C with $(prev\_token + num\_regex + next\_token)$
    $O \leftarrow$ LLM $M(O + g_i, guide_{num}, cache)$
  end if
  Update cache
end for
$O \leftarrow O + g_{\#literals}$
return O

4.1.1 Template with synthetic queries. To simulate the workload in real enterprise databases where repeated query templates are commonplace (as we show in Section 5.1), we augment each of these benchmarks as follows. Let T be a set of labeled Text-SQL pairs in a schema. We first split T into two equal halves randomly called matched T_m and unmatched set, $T - T_m$.

For each Text $\mathbf{x}$ and SQL $\mathbf{y}$ in the matched set T_m, we generate one synthetic SQL $\mathbf{y}'$ and its corresponding NLQ $\mathbf{x}'$ using OpenAI o3-mini. The synthetic SQL $\mathbf{y}'$ samples a different value of literal from the database compared to what is present in the original SQL $\mathbf{y}$. The corresponding NLQ $\mathbf{x}'$ is a natural language utterance that the LLM provides of $\mathbf{y}'$. We further prompt the LLM to ensure that $\mathbf{x}'$ is a sufficiently different paraphrase of $\mathbf{x}$.

Now the labeled pool D available for indexing is the pairs $(\mathbf{x}', \mathbf{y}')$, whereas the original pairs $(\mathbf{x}, \mathbf{y})$ are used to evaluate the LLM. Thus, the whole of T is used for evaluation, with the subset in the matched set guaranteed to find a matching template in D.

4.1.2 Non-synthesized Template and Test Set. To provide a more comprehensive understanding of TeCoD's practical performance and address concerns regarding potential biases introduced by this synthetic data augmentation, we conducted an evaluation on a test set and template pool created entirely out of real queries in the benchmark. We found only a small number of recurring SQL templates in this workload. In the BIRD-dev set, we found 61 templates covering 134 queries out of 1534, and in the Spider-dev set 482 templates covering 966 questions out of 1034. We evaluate our system with this subset of 61 and 482 queries as our template pool, and use it to create our template bank along with paraphrases generated on these questions. The rest of the questions that

are covered by the template make up our test set. Our test set on Bird is comprised of 73 matching queries across databases, and we sampled an equal number of non-matching queries to make the split even, making it a total of 134 queries. On Spider, we have a test set of 499 queries, of which 484 are matching and 15 are non-matching queries. The split is not even in the case of Spider, as most of the queries are covered by the template, and a few databases have no templates, and similarly for Bird, one database has no templateable questions. Please refer Table 10 in Appendix A.1 for more details.

Evaluation Metrics. We use a widely adopted metric, Execution Match Accuracy (ExM). The ExM metric evaluates whether the predicted SQL and the gold SQL yield the same execution results on the database. Further, we use **Inference Latency**, to measure the efficiency of the decoding pipeline and the throughput improvements over the baseline.

4.2 Large Language Models

We evaluate our system on general purpose models like Llama-3.1-8B-Instruct [12] and Granite-3.1-8B-Instruct [11], and supervised fine-tuned models from the CodeS [18] series with 1B and 15B parameter variants, CodeS-1B-Bird-with-evidence, CodeS-1B-Spider, CodeS-15B-Bird-with-evidence, and CodeS-15B-Spider. We extend our evaluation to include recent SOTA models for the Text2SQL task such as XiyanSQL Qwen- coder (7B and 14B) [21] and Arctic-R1-7B [34], which are publicly available, provide SOTA results on BIRD Single-Model Leaderboard, and comparable in scale to the models used in our current experiments. Of these Arctic-R1-7B is a reasoning model and is almost an order of magnitude slower than the rest of the models. We will see that our method is orthogonal to the baseline model used, and provides gains across all LLMs. We did not include other recent models like DIN-SQL [24] or CHASE-SQL [23] because either the code is not publicly available or we included others like QwenCoder which are established better.

SQL Generation using LLM. We prompt the LLM for generating SQL using an LLM specific prompt. In each case, there is a generic natural language instruction, followed by a description of the schema and metadata of the database queried, followed by the in-context examples, and then the current test question **q**. However, instead of providing the entire database metadata, we filter the meta data as described below.

Schema Filtering. Enterprise databases are comprised of a large number of tables and columns per table. If we include complete schema in the prompt, it will exceed the model's context length. We use the models developed by CodeS [18] to do schema subsetting. Essentially, at inference time, we input database schema and NLQ to get relevance scores and using those scores, we choose top-k tables and columns, which would become part of the schema prompt.

Value Retriever. In order to inform the LLM of the values of categorical columns that might possible match user NLQ, it is necessary to also retrieve candidate matching values from the database. Suppose the question is "Who is the last F1 winner of the Sepang GP?" there has to be a correct linking between "Sepang" and the city column of the table that contains the winner's data. Here, again, we reuse the models in the CodeS repository to retrieve values. They proposed a coarse-to-fine-grained matching approach, where first, they use the BM25 index for the initial search and later use the longest common subsequence (LCS) algorithm to find the most relevant values.

Grammar Constrained Decoding. For constraining the output of the LLMs, we are using Outlines [32] Python library.

4.3 Training the Template Matcher

We fine-tuned the distilbert-base-uncased [26] model for the NLI task using the training split of the BIRD benchmark (See A.3 for the details on the embedding and NLI models used with non-synthesized workload). We generated 20 alternate NLQs per question from the train set using OpenAI o3-mini to serve as positive pairs. To train the model, we needed hard negatives, which we mined using cosine similarity and added an equal number of negative pairs (with unmasked NLQ). Also, we added the same number of positive and negative pairs with masked NLQ to make the model more robust. Note, that the template matcher model database-agnostic and is shared across databases unlike the template index. The databases from which queries are sampled during training are disjoint from the databases used during testing. The model is able to generalize because the task is much simpler — establishing the semantic correspondence between user NLQ and masked-NLQs (from $\mathcal{I}$). The BIRD train provides a large number of examples. Training on BIRD's train set does not give an unfair advantage on BIRD dev set because the databases in these splits are entirely different.

For each database schema we create a different template index since our template refer to table and column names of a schema. Our system is designed for harnessing the previously seen queries of a specific enterprise database, to improve SQL generation performance of frequent queries.

5 Results

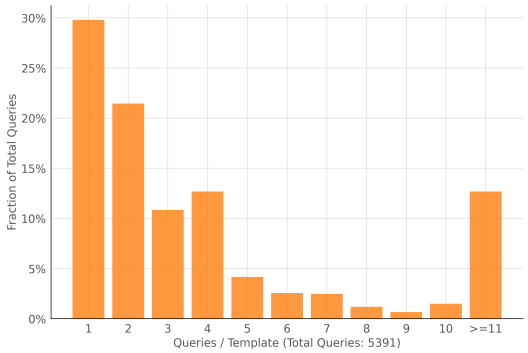


Fig. 4. Workload distribution of a large bank. The X-axis shows the size of each template in terms of number of queries in that template. The Y-axis shows the fraction of total queries that are present in templates of that size. This chart shows that 30% of the queries are in templates of size 1, and 22% in templates of size 2. We show that more than 50% of the queries would find a matching template in prior queries if queries arrive sequentially in the above workload.

5.1 Role of Templates in Real workloads

As discussed earlier, standard benchmarks like BIRD and SPIDER do not preserve the occurrence frequency of repeated queries that is typical in large enterprises. We were able to access the proprietary workload of a large bank in a large country. The workload reflects natural language queries submitted by business executives to the DBAs of the main data warehouse of the bank. Bank queries are OLAP in nature. Most of the queries are for analysis, reporting, or business intelligence purposes. We grouped queries based on whether they follow the same parametric template. Each group defines a template, and we define the size of a template as the number of queries that fall in that group. In Figure 4 we show for each template size, the fraction of the total query workload

that are part of a template of that size. Observe that 30% of the queries in the workload belong to a template of size 1, and about 22% to templates of size 2 etc. From these numbers, we can estimate that if queries in the workload were to arrive sequentially then for more than 50% of the queries, a previous query with a matching template would have been observed. In this case, every generated SQL is manually inspected by the DBA, so all previously occurring queries would be paired with a verified correct SQL. With our system for more than 50% of the queries, the matched template could be used to automatically generate a correct SQL with significantly higher accuracy than a baseline Text-to-SQL system.

TeCoD is built on the premise that while natural language phrasing is diverse, the underlying SQL structure for common business queries is often recurrent. While the effectiveness of our system relies on having coverage for a given query in the template index, it is specifically designed to target these high-frequency head queries. As demonstrated by the enterprise workload analysis, where over 50% of queries could find a matching template upon sequential arrival, TeCOD provides a pragmatic path towards high-accuracy SQL generation for the most common analytical needs, thereby significantly enhancing reliability for the bulk of an organization's day-to-day data interaction.

5.2 Overall Text-to-SQL Execution Accuracy

In Table 5 we compare the execution accuracy of the following methods on eight LLMs of different sizes and recency:

(1) ZeroShot: Where no in-context examples are provided in the prompt.
(2) ICL-3: Where three labeled Text-SQL pairs from D whose NLQ is most similar to $\mathbf{q}$.
(3) TeCoD-SGC: Here we match templates using TeCoD, but instead of hard enforcement of the matched templates using GCD, we prompt with an additional instruction to the LLM to generate the SQL for $\mathbf{q}$ in accordance with the template $\tilde{\mathbf{y}}$.
(4) TeCoD: Here we follow the full pipeline as outlined in Figure 2.

We did not consider fine-tuning with the queries used for template construction because the number of queries is small. Importantly, our setting assumes templates arrive over time, not all at once. Fine-tuning would require repeated retraining, which is impractical. Also with limited data, fine-tuning risks overfitting, forgetting, and other side effects. Moreover, FT is not always feasible in multi-tenancy models.

In Table 5 we include results of all four datasets as described. However, comparisons across datasets is not meaningful since the Synthetic and Non-Synthetic versions have very different test-sets as seen in Table 10. Here the workload comprises of only queries for which a matching template exists in D, however, the identity of the matching template needs to discovered. We present results over a mix of matched and unmatched queries in Section 5.3.

In Table 5 comparing among the four methods across dataset-LLM combination, we can make these interesting observations.

(1) Zero-shot accuracy is quite poor on the BIRD benchmark across all LLMs. The Spider benchmark is considered easier, and there ZeroShot accuracy is much higher.
(2) In-context examples provide improvements for all dataset-LLM combinations. Note, the selected ICL examples likely include one example with the matching template, and even then the accuracy boost is modest for most LLMs.
(3) Once TeCoD's template selection module chooses a template, and we instruct the LLM to follow the template (TeCoD-SGC method), we observe a huge jump in accuracy beyond ICL-3.
(4) However, TeCoD-GCD with strict enforcement of the chosen template provides significantly greatest overall gains, particularly for smaller models like CodeS-1B and Granite-2B. In most cases, we achieve more than 89% accuracy, and there is a huge jump in accuracy over TeCoD-SGC.

Table 5. BIRD and Spider ExM% accuracy for various methods on various LLMs (CodeS-Bird-with-evidence for the BIRD and CodeS-Spider for the Spider dataset). Our method (TeCoD-GCD) that performs template constrained decoding provides significantly higher accuracy than the next best method ICL-3, for both synthesized and non-synthesized matched sets.

Model	Method	Syn		Non-Syn	
		BIRD	Spider	BIRD	Spider
CodeS-1B	ZeroShot	40.91	70.33	63.01	72.11
	ICL-3	50.65	79.37	68.49	78.10
	TeCoD-SGC	54.55	82.32	64.38	79.75
	TeCoD-GCD	86.88	98.04	83.56	95.04
Granite-3.1	ZeroShot	26.23	64.44	43.84	66.74
2B-Instruct	ICL-3	62.99	86.44	71.23	82.23
	TeCoD-SGC	74.94	94.70	79.45	89.46
	TeCoD-GCD	86.88	98.43	84.93	95.87
Llama-3.1	ZeroShot	44.81	77.01	61.64	75.00
8B-Instruct	ICL-3	73.90	93.52	83.56	89.46
	TeCoD-SGC	84.16	94.30	87.67	91.32
	TeCoD-GCD	89.22	96.27	89.04	94.01
Granite-3.1	ZeroShot	40.26	71.12	61.64	73.14
8B-Instruct	ICL-3	62.47	87.03	84.93	84.09
	TeCoD-SGC	78.31	89.98	87.67	86.98
	TeCoD-GCD	89.22	98.62	89.04	95.45
CodeS-15B	ZeroShot	51.30	81.34	68.49	81.40
	ICL-3	64.29	88.02	78.08	87.40
	TeCoD-SGC	69.35	88.61	78.08	86.98
	TeCoD-GCD	88.70	97.25	90.41	94.63
SOTA LLMs for Text-to-SQL					
XiYanSQL	ZeroShot	54.10	90.37	75.34	90.70
QwenCoder	ICL-3	77.79	95.48	84.93	94.63
7B-2504	TeCoD-SGC	81.95	94.50	89.04	92.98
	TeCoD-GCD	89.48	97.45	87.67	95.66
XiYanSQL	ZeroShot	55.66	94.30	73.97	93.80
QwenCoder	ICL-3	79.87	97.45	89.04	96.49
14B-2504	TeCoD-SGC	90.39	98.43	93.15	97.31
	TeCoD-GCD	88.05	99.02	90.41	96.07
Snowflake	ZeroShot	63.25	89.00	79.45	88.02
Arctic	ICL-3	69.22	93.52	79.45	92.15
Text2SQL	TeCoD-SGC	75.32	94.70	83.56	92.77
R1-7B	TeCoD-GCD	86.36	97.05	89.04	93.60

(5) As seen in the bottom part of the table, recent LLMs specifically trained for SQL generation provide much higher zero-Shot and few-shot accuracy than earlier models, but even on these TeCoD is able to provide gains. While TeCoD-GCD generally demonstrates superior accuracy for matched queries compared to TeCoD-SGC across various LLMs and datasets, a nuanced observation

arises when evaluating newer, state-of-the-art models on this data. For highly capable LLMs like XiYanSQL QwenCoder-14B, which possess robust inherent understanding and SQL generation capabilities, the "soft guidance" provided by TeCoD-SGC may suffice. These advanced models might be proficient enough to adhere to the template without strict grammar enforcement, potentially remedying the errors of template selection. However, the latency of these newer models could be prohibitive, with latencies of 6.94s for XiYanSQL QwenCoder-7B and 9.67s for XiYanSQL QwenCoder-14B. In contrast, TeCoD-GCD delivers close enough accuracy on Granite-8B, which is significantly more responsive with a latency of just 3.64s.

In summary, TeCoD not only elevates weaker and smaller base models (like Granite-3.1-8B-Inst) to match performance of larger, stronger baselines (e.g., QwenCoder-14B ICL-3), but it also improves the performance of larger models. This positions TeCoD as a generalizable and impactful inference enhancement method for Text-to-SQL tasks.

Error Analysis. While TeCoD achieves strong execution accuracy across datasets, there remain a small number of cases (8-12%) where the generated SQL is incorrect. We present a brief analysis of the reasons for these errors. First, we account for the errors due to not being able to identify the correct template. Second, the bulk of the error is due to the wrong literal generation even after matching with the correct template. We analyze the nature of these errors.

(1) A common source of error is when the literal name is an obscure entry (e.g., domain-specific jargon or an internal abbreviation) in the database, and the schema subset shown to the LLM prompt fails to retrieve the literal name.

(2) Another source of errors is number literals arising out of LLMs difficulty with numerical reasoning. An example is shown below:

```
Gold: SELECT T1.frequency, T2.k_symbol ... WHERE T1.account_id = 3 AND T2.
    total_amount=3539
Pred: SELECT T1.frequency, T2.k_symbol ... WHERE T1.account_id = 3539 AND T2.
    total_amount=3539
```

(3) A third source of errors is failure of grammar constrained decoding to terminate string literals particularly when the string itself contains a quote. Such behavior is especially noticeable in queries containing non-ASCII characters in string literals.

```
str_regex: '([^']|'')*'
Gold: SELECT type FROM sets WHERE code IN ( SELECT setCode FROM
    set_translations WHERE translation='Huitième
    'edition' )
Pred: SELECT type FROM sets WHERE code IN ( SELECT setCode FROM
    set_translations WHERE translation='Huitième édition'') GROUP BY TYPE;
    SELECT TYPE FROM sets WHERE code = 'n' )
```

Here, after generating tokens for the string literal, the LLM outputs `''` instead of `'` or `')`, and the grammar considers it as escaped single quote. As a result, the LLM can continue generating any token permitted by the string regex which includes all possible characters until stopping criteria is reached.

We present more examples of errors in Section A.2 of the Appendix.

5.3 Impact of TeCoD on queries without matching templates

In TeCoD, when the template matcher incorrectly identifies a template for an unmatched NLQ, accuracy of those queries drop. We observed that zero-shot accuracy is the same roughly across

Table 6. Accuracy of template selection and rejection on BIRD and Spider datasets for baseline, our method, and various ablations on our method. Best accuracy provided by our NLI-based template matching module without masking the user question. Baseline method based on cosine similarity of question and template embeddings provides much worse matches.

Dataset	BIRD			Spider		
Method	Selection	Rejection	Average	Selection	Rejection	Average
Baseline	63.51	82.85	73.14	74.66	75.24	74.95
Ours, No mask	92.21	89.01	**90.61**	97.64	86.86	92.17
Ours, With Masking	91.82	88.87	90.35	97.84	87.05	**92.36**
Ours Gold Masking	91.69	88.61	90.16	97.64	86.86	92.17
Ours, 1 NLQ	87.14	87.43	87.29	92.93	84.38	88.59
Ours, No FT	68.31	83.77	76.01	87.23	82.29	84.72
Ours, No FT, No mask	30.91	95.29	62.97	56.39	89.33	73.11
Ours, No NLI	71.43	89.66	80.51	92.34	85.71	88.97

matched (M) and unmatched (U). ICL causes accuracy to jump by almost 20% on an average for the matched set (M), while providing only a modest 2% gains for the unmatched set. With TeCoD, accuracy drops by between 3–7% for the unmatched set, but because of the dramatic jump in accuracy of the matched set, the overall average accuracy improves. We observe jumps by 15-36% for BIRD-dev over ICL-3 and 4-18% jump for Spider-dev across LLMs. In addition to the improved accuracy we also observe 2.2x lower latency. TeCoD-GCD is particularly useful in reducing the latency of reasoning models like Arctic Text2SQL where the baseline model takes 20 seconds per query on average, which reduces to 6 seconds. Even though with TeCoD, accuracy drops by a small amount for the unmatched set, but since the baseline accuracy for this set is already low, (average accuracy of 50% for BIRD), the practical impact of such a drop may be limited. Also, as an enterprise collects more queries to the template pool, the unmatched pool size is expected to shrink.

5.4 Template Selection Accuracy

We next present the efficacy of our template selection module described in Section 3.2. To bring out the merit of different design decisions in that module, we present a comparison with a number of ablations and baseline. In each case, we measure two kinds of accuracy: (1) Selection accuracy for queries where a matching template is known to be present in $\mathcal{I}$. Errors in this case could be either because of deciding that no matching template is present, or choosing the wrong template. (2) Rejection accuracy for queries where no matching template is present. For such queries, the correct output is to reject all templates in $\mathcal{I}$. We have an equal number of matched and unmatched queries, and also present Overall accuracy as the average of the two.

Table 6 compares the following methods:

(1) Baseline: A baseline method for template matching is to compare the embedding of the template with that of the NLQ using established methods like Cosine similarity. Let $\tilde{\mathbf{y}}$ be a candidate template, and we need to decide if a user question $\mathbf{q}$ would lead to an SQL with template $\tilde{\mathbf{y}}$. In this method, this decision is made based on whether Cosine similarity of the embedding of $\tilde{\mathbf{x}}, \mathbf{q}$ is greater than a threshold η. To calculate the threshold, we create a sample set of questions with an equal number of positives and negatives by choosing a positive and a negative example per question from the synthetically generated Text-SQL pairs. The threshold is chosen to maximize the difference between the true positive rate and the false positive rate, giving the best balance between true positives and false positives.

(2) TeCoD's template matcher as described in Section 3.2 where we use a fine-tuned NLI model. Details about NLI model training appear in Section 4.3. We use the same model for Spider and BIRD. Observe that our matching module provides significantly higher accuracy for both selection and rejection of templates. compared to baseline of 73.14%, we achieve an accuracy of 90.61%.
(3) Ours, with Masking: In this version, we also mask literals in each arriving user query **q** by first generating an SQL using the default LLM, and then fuzzy masking the literals in the SQL from **q**. This method of masking the literal in the user question entails an additional overhead of generating the SQL using a default method. In any case, there is no guarantee that the generated SQL is correct. We observe that the accuracy stays more or less the same compared to our default no-masking approach. One reason for this robustness is that we trained the NLI model with a mix of masked and unmasked user questions.
(4) Ours with Gold Masking: In order to firmly establish the role of masking, we consider an oracle setting, where we use the gold SQL of the user NLQ **q** to extract the literals, and mask them in the user question. Masking with gold literals improves accuracy, but only slightly. Based on these experiments we resolved to not mask literals in TeCoD's final pipeline.
(5) Ours, Single NLQ per template: We next study the impact of including multiple synthetic NLQs with template. With just a single NLQ per template, the accuracy drops by almost 3.32% drop compared to with 10 NLQs.
(6) Ours with untuned NLI model: When using the untuned NLI model (we used HF tasksource/deberta-base-long-nli) accuracy dropped significantly from 90.35% to 76.01%.
(7) Ours with untuned NLI model for unmasked NLQ: When the untuned NLI model is applied on umasked user questions, the drop in selection accuracy is drastic going from 92% to 31%. This shows that our strategy of fine-tuning the NLI model with a mix of masked and unmasked user question was essential to enable the NLI model to perform well even on unmasked user questions.

Table 7. Execution Match(EX) and inference latency for different methods of template constrained generation on BIRD and Spider dev sets. For reference we also show the latency of unconstrained generation. Our method TeCoD provides almost the same accuracy as using the full Flexible Template constraints, while reducing running time by about half compared to library default. CodeS-1B/15B is used for both BIRD and Spider, but refers to separate LLMs finetuned on each dataset's training set.

Method	BIRD							
	CodeS-1B		CodeS-15B		Llama-3.1 8B Instruct		Granite-3.1 2B Instruct	
	EX (%)	Latency	EX (%)	Latency	EX (%)	Latency	EX (%)	Latency
Unconstrained	38.07	1.00×	48.57	1.00×	39.18	1.00×	24.12	1.00×
TeCoD	91.40	1.15×	92.44	0.87×	92.18	1.24×	88.07	0.62×
TeCoD (No Two-phase decoding)	91.85	2.66×	92.89	1.34×	92.50	2.74×	88.33	1.44×
TeCoD (No context)	84.22	1.83×	85.59	1.21×	79.47	1.59×	73.53	1.60×
Fixed Template (No context)	69.17	2.14×	71.97	1.45×	35.66	1.83×	62.52	2.16×
Fixed Template (Left and right context)	91.20	1.15×	92.57	0.85×	92.11	1.13×	85.46	0.63×
	Spider							
Unconstrained	70.50	1.00×	80.08	1.00×	73.69	1.00×	65.38	1.00×
TeCoD	98.94	0.32×	96.13	0.29×	99.23	0.43×	99.03	0.16×
TeCoD (No Two-phase decoding)	99.03	2.30×	96.13	1.28×	99.23	2.83×	99.03	1.22×
Fixed Template (Left and right context)	99.13	0.33×	99.13	0.29×	99.13	0.44×	99.03	0.16×

Table 8. Robustness of template grammars compared to Baseline on BIRD dev set. Fixed Template is followed for all these methods where Gold SQL in modified in different ways such as converting to small case, replacing single whitespace with random whitespaces, pretty-printing the template grammar. Significant drop is observed in ExM for all models except Granite.

Gold SQL Modification	CodeS-1B	CodeS-15B	Llama-3.1-8B	Granite-3.1-2B
Fixed Template	91.20	92.57	92.11	85.46
Small Case SQL	90.61	92.44	90.81	86.83
Pretty Format	91.40	92.76	88.92	86.31
Random Spaces (2,3)	34.88	69.23	79.99	88.14
Random Spaces (2,5)	32.07	64.99	57.50	83.70

5.5 Template Constrained Decoding

Grammar-constrained decoding is essential for ensuring the syntactic validity of SQL queries generated by LLMs. However, existing methods often suffer from high latency and are sensitive to formatting variations. In this section, we evaluate TeCoD, our proposed method that aims to balance correctness and efficiency. We benchmark all methods on the BIRD and Spider dev sets, which contain 1534 and 1034 queries respectively. Table 7 presents performance comparisons across different decoding strategies:

(1) **Unconstrained:** This is the standard decoding setup where the SQL is generated without any template constraints. As expected, unconstrained generation is generally faster than grammar constrained methods. However, this comes at the cost of significantly lower execution accuracy (ExM).

(2) **TeCoD**: We next compare the constrained generation algorithm of TeCoD where we use the flexible grammar with the efficient two-phase partitioning decoding algorithm 3. We observe huge jump in accuracy with constrained decoding without incurring latency overheads. On the BIRD dataset, TeCoD achieves latency comparable to — and for some LLMs (CodeS15B and Granite), even better than — unconstrained decoding. On Spider, latency improvements range from 2 - 6× over the unconstrained method. TeCoD's accuracy and efficiency gains are due to important design decisions, and we present an ablation on each of these next.

(3) **TeCoD Without Two-phase Decode**: If we run the flexible grammar with the default GCD algorithm 2 instead of the two phase efficient inference of algorithm 3, we incur a factor of two to three times latency overhead compared to unconstrained generation. This shows that off-the-shelf GCD methods are expensive. The slight accuracy drop arises because TeCoD supplies the entire partition in a single generate call, which alters the logit distribution due to layer normalization effects.

(4) **TeCoD without Context Tokens**: Another crucial factor for accurate generation with partitioned decoding was to include left and right context tokens to adjust for LLM's tokenization that can straddle across partition boundaries. We oberve that dropping the context tokens causes huge drop in accuracy. On the BIRD dataset, accuracy drops from above 90% to around 80% when averaged across LLMs. For some LLMs, example Llama the drop is huge — from 92% to 79%. We present some anecdotes:

```
Examples:
1. SELECT T2.`School Name` ... `Enrollment (K-12)` > 0.1 AND NumGE1500 > 0
2. SELECT Website FROM schools ... AdmLName1 = 'Larson') OR (AdmFName1 = '
    Dante' AND AdmLName1 = 'Alvarez')
```

In example 1, if the next token " AND" is not provided as part of the grammar when generating the literal 0.1, the LLM may repeatedly generate numbers instead of predicting an <eos> token. Similarly, in example 2, omitting the ')' token from the grammar while generating 'Larson'could lead to malformed outputs. This is because missing the next token ID results in over-masking of valid token completions.

(5) **Fixed Template:** Next we establish the usefulness of the flexible grammar by replacing with the Fixed template, an example of which is shown in the second row of Table 3. We report accuracy of this grammar too both with and without the context tokens. We observe that without the context tokens, the fixed template method shows much bigger accuracy drops. When we extend the fixed grammar with left and right context tokens, the accuracy does bounce back to be almost comparable to what we obtained with the flexible grammar.

Across multiple models and datasets (BIRD and Spider), the two-phase decoding approach shows 1.5× to 2.3× improvements in latency over the vanilla constrained decoding method (referring to rows "TeCoD" and "TeCoD(No Two-phase decoding)" in Table 7 for both BIRD and Spider. These results demonstrate that our method reduces unnecessary computation and improves efficiency during inference. Overall, it may appear that Flexible Template does not provide much gains beyond Fixed Template. That may be because current LLMs are already exposed to the SQL formatting deployed in public Text-to-SQL benchmarks like BIRD or Spider. We show that the accuracy of Fixed Template is highly sensitive to the surface formatting of the SQL template. Table 8 shows results for various perturbations: converting only SQL keywords to lowercase (literal values remain case-sensitive), randomly replacing single spaces between keywords with 2–5 or 2–3 spaces, and Pretty-Format where we reformat the SQL using standard indentation and line breaks using SQLGlot to improve readability. CodeS and LlaMA models show sharp accuracy drops with these formatting changes, whereas Granite remains more robust but the best accuracy of the Granite model is lower. These findings reinforce that fixed templates created out of the given SQL are highly sensitive to formatting of $\tilde{\mathbf{y}}$. The accuracy of TeCoD is invariant to the string form of the template because it expresses all templates using a flexible grammar (shown in the last row of Table 3).

6 Conclusion and Future Work

Conclusion. We introduced TeCoD, a system designed to significantly boost Text-to-SQL accuracy by utilising templates derived from frequently occurring queries. TeCoD converts labelled Text-to-SQL pairs into reusable templates and employs an accurate matching model to identify user queries that conform to these templates. For matched queries, TeCoD uses a template-constrained decoding process, achieving substantial accuracy improvements (up to 36% over ICL) and enhanced inference latency (1.5-2.2x faster).

Most significantly, the paper establishes that for enterprise workloads with recurring query patterns, which can represent more than 50% of queries in real-world settings, TeCoD provides a practical path to high-accuracy Text-to-SQL conversion. This approach is particularly valuable as it works without fine-tuning and its design for incremental deployment, making it particularly suitable for enterprise environments where workloads evolve. The system's accuracy and coverage are not static; as new, expert-verified NLQ-SQL pairs become available, they can be seamlessly converted into templates and added to the template pool, allowing the system to continuously adapt and improve its performance. Crucially, TeCoD provides a robust fallback mechanism. When no matching template is found, the system defaults to a standard generation method using in-context examples, ensuring that it can service both frequent, recurring queries and novel, ad-hoc ones.

The extensive evaluations on both the BIRD and Spider benchmarks across different LLMs consistently demonstrate that TeCoD outperforms both zero-shot and in-context learning approaches,

often by substantial margins. The error analysis provides crucial insights into remaining challenges as to why TeCoD generated incorrect SQL in a small number of cases.

Future Work. The current implementation focuses on templates where only literals are masked. Future research could explore more flexible template representations. While TeCoD excels at handling head queries, tail queries still rely on standard in-context learning. Exploring hybrid approaches that combine elements of template-based generation with more flexible generative techniques could help improve performance across the entire query distribution. The principles of identifying recurring patterns and enforcing structural constraints during generation are likely to be broadly applicable.

Acknowledgments

We acknowledge the support of the SBI Foundation Hub for Data Science & Analytics at the Indian Institute of Technology Bombay for providing financial support and infrastructure for conducting the research presented in this paper.

A Appendix

A.1 Analysis of NL similarity and template match success

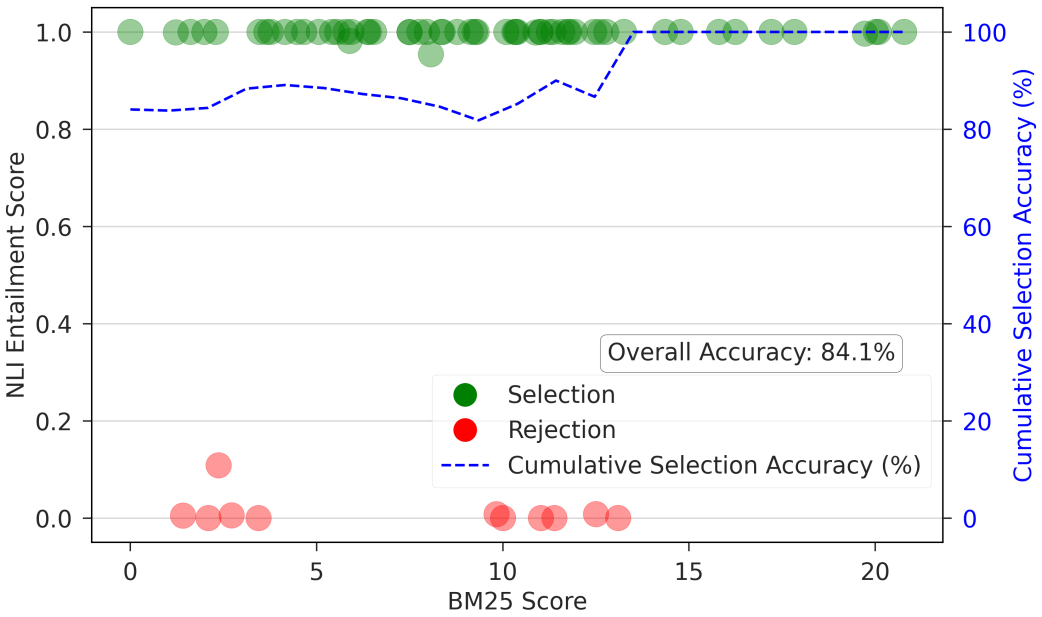


Fig. 5. Scatter plot of BM25 score of the query with its most similar alternate NLQ from the matching template over Bird-Non-synthesized matched test set.

We analyze how similar a query has to be with a stored template for the template matcher to work. For this, in Figure 5 we present a scatter plot of the BM25 similarity of the query with its template against the match probability output by the template matcher. We observe that cumulative selection accuracy increases as expected as BM25 similarity increases, but even at low similarity levels, we get accuracy close to 80%. also present a few examples where the template matcher selected and rejected the template in Table 9.

Table 9. Examples of template matcher performance, showing cases of successful and failed matches based on BM25 similarity.

Outcome	BM25 Score	Query	Template
		Rejection Cases	
Rejection	2.10	Please list the leagues from Germany.	Tell me the name of the country's football league for _.
Rejection	2.37	Give the number of "Revival" badges.	What is the tally of users who obtained the _ badge?
		Selection Cases	
Selection	1.22	In the non-carcinogenic molecules, how many contain chlorine atoms?	Retrieve the total number of molecules known to cause _ that have _ in their makeup.
Selection	1.62	What is Abomination's superpower?	Give me the super abilities that _ is known for.

Table 10. Dataset Statistics

Dataset	#Databases	Matched	Unmatched	Template
BIRD-Syn	11	0	764	770
Spider-Syn	20	0	525	509
BIRD-Real	10	73	61	73
Spider-Real	16	484	15	482

A.2 Analysis of literal generation errors

(1) Templates using a particular SQL syntax were consistently associated with incorrect literal generation. Since all queries with this syntax resulted in errors, it is plausible that the LLM had limited or no exposure to such patterns in its training data.

```
Example:
Gold: SELECT CAST(`Free Meal Count (K-12)` AS REAL) / `Enrollment (K-12)`
    FROM frpm ORDER BY `Enrollment (K-12)` DESC LIMIT 9, 2
Pred: SELECT cast( "Free Meal Count (K-12)" AS REAL ) / "Enrollment (K-12)
    " FROM frpm ORDER BY "Enrollment (K-12)" DESC LIMIT 10 OFFSET 9
```

(2) In some cases, the schema item and its correct literal value are included in the prompt, yet the LLM generates a slightly different version of the literal, such as `'='` being produced as `' = '`. In such cases often disagreement is observed between schema item value presented by schema and evidence in prompt.

```
Example:
Prompt: database schema :
table bond , columns = [ bond.bond_type ( text  values : - , = ) , bond.
    molecule_id ( text  values : TR000 , TR001 ) , bond.bond_id ( text
    primary key  values : TR000_1_2 , TR000_2_3 ) ] ...
```

```
Evidence:double bond refers to bond_type = ' = '
Question: Please list top five molecules that have double bonds in
    alphabetical order.
Gold:SELECT DISTINCT T.molecule_id FROM bond AS T WHERE T.bond_type = '='
    ORDER BY T.molecule_id LIMIT 5
Pred:SELECT DISTINCT T.molecule_id FROM bond AS T WHERE T.bond_type = ' =
    ' ORDER BY T.molecule_id LIMIT 5
```

A possible explanation is the recency bias often observed in LLMs, since the evidence snippet (with extra whitespace) appears closer to the end of the prompt.

(3) When generating format specifiers for dates, the LLM often produces incorrect literals.

```
Example:
Gold: SELECT DISTINCT T1.ID, STRFTIME('%Y', CURRENT_TIMESTAMP) - STRFTIME
    ('%Y', T1.Birthday) FROM Patient AS T1 INNER JOIN Examination AS T2 ON
     T1.ID = T2.ID WHERE T2.RVVT = '+'
Pred: SELECT DISTINCT T1.ID, strftime('%J', CURRENT_TIMESTAMP) - strftime(
    '%J', T1.Birthday) FROM Patient AS T1 INNER JOIN Examination AS T2 ON
    T1.ID = T2.ID WHERE T2.RVVT = '+'
```

(4) On the Spider dataset, CodeS-15B-Spider shows a drop in Execution Match (ExM) due to consistent errors in string literal generation. The model overfits to formatting with two spaces between the comparison operator and the string literal, failing when only one space is allowed likely due to exposure to such formatting in the training data. When evaluation permits variable spacing, it generates correct literals, indicating the issue stems from overfitting to surface formatting rather than misunderstanding the query.

A.3 Embedding and NLI model

As detailed in the paper, our template matching module initially utilized the NV-Embed-v2 model for creating embeddings and a fine-tuned distilbert-base-uncased model for the Natural Language Inference (NLI). While evaluating for the non-synthetic workload, we have updated these components to use more recent, state-of-the-art models. Specifically, we now employ Qwen/Qwen3-Embedding-4B [39] to generate embeddings and a fine-tuned Qwen/Qwen3-Reranker-4B [39] for the NLI classification task. The core methodology is unchanged: the embedding model is used directly to perform an initial similarity search for candidate retrieval, and the reranker is subsequently fine-tuned and used as the NLI model to validate and select the final template. However, we found using the mean of NLI scores of alternates from the same template to be better and used the same for selecting the template.

References

[1] Abhijeet Awasthi, Ashutosh Sathe, and Sunita Sarawagi. 2022. Diverse Parallel Data Synthesis for Cross-Database Adaptation of Text-to-SQL Parsers.

[2] Max Bachmann. 2024. *rapidfuzz/RapidFuzz: Release 3.8.1.* doi:10.5281/zenodo.10938887

[3] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2024. Guiding LLMs The Right Way: Fast, Non-Invasive Constrained Generation. arXiv:2403.06988 [cs.LG] https://arxiv.org/abs/2403.06988

[4] Adithya Bhaskar, Tushar Tomar, Ashutosh Sathe, and Sunita Sarawagi. 2023. Benchmarking and Improving Text-to-SQL Generation under Ambiguity. In *The 2023 Conference on Empirical Methods in Natural Language Processing.* https://openreview.net/forum?id=a0yFO9gKc5

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL] https://arxiv.org/abs/1810.04805

[7] Ju Fan, Zihui Gu, Songyue Zhang, Yuxin Zhang, Zui Chen, Lei Cao, Guoliang Li, Samuel Madden, Xiaoyong Du, and Nan Tang. 2024. Combining small language models and large language models for zero-shot NL2SQL. *Proceedings of the VLDB Endowment* 17, 11 (2024), 2750–2763.

[8] Han Fu, Chang Liu, Bin Wu, Feifei Li, Jian Tan, and Jianling Sun. 2023. Catsql: Towards real world natural language to sql applications. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1534–1547.

[9] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. Text-to-sql empowered by large language models: A benchmark evaluation. *arXiv preprint arXiv:2308.15363* (2023).

[10] Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. 2023. Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore. https://aclanthology.org/2023.emnlp-main.674

[11] Granite Team and IBM. 2024. Granite-3.1-8B-Instruct. https://huggingface.co/ibm-granite/granite-3.1-8b-instruct. https://huggingface.co/ibm-granite/granite-3.1-8b-instruct Model card release date: December 18, 2024.

[12] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[13] Mayank Kothyari, Dhruva Dhingra, Sunita Sarawagi, and Soumen Chakrabarti. 2023. CRUSH4SQL: Collective Retrieval Using Schema Hallucination For Text2SQL. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 14054–14066. doi:10.18653/v1/2023.emnlp-main.868

[14] Chankyu Lee, Rajarshi Roy, Mengyao Xu, Jonathan Raiman, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. 2024. NV-Embed: Improved Techniques for Training LLMs as Generalist Embedding Models. *arXiv preprint arXiv:2405.17428* (2024).

[15] Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2024. Mcs-sql: Leveraging multiple prompts and multiple-choice selection for text-to-sql generation. *arXiv preprint arXiv:2405.07467* (2024).

[16] Fei Li and H. V. Jagadish. 2014. Constructing an interactive natural language interface for relational databases. *Proc. VLDB Endow.* 8, 1 (Sept. 2014), 73–84. doi:10.14778/2735461.2735468

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

[18] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024. CodeS: Towards Building Open-source Language Models for Text-to-SQL. arXiv:2402.16347 [cs.CL]

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

[20] Yunyao Li and Davood Rafiei. 2017. Natural Language Data Management and Interfaces: Recent Development and Open Challenges. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3035918.3054783

[21] Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li, Bolin Ding, and Jingren Zhou. 2025. XiYan-SQL: A Novel Multi-Generator Framework For Text-to-SQL. (2025). arXiv:2507.04701 [cs.CL] https://arxiv.org/abs/2507.04701

[22] Toby Mao and contributors. [n. d.]. SQLGlot: Python SQL Parser, Transpiler, and Optimizer. https://github.com/tobymao/sqlglot. Accessed: 2025-01-24.

[23] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan Ö. Arik. 2024. CHASE-SQL: Multi-Path Reasoning and Preference Optimized Candidate Selection in Text-to-SQL. *ArXiv* abs/2410.01943 (2024). https://api.semanticscholar.org/CorpusID:273098638

[24] Mohammad Reza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. *ArXiv* abs/2304.11015 (2023). https://api.semanticscholar.org/CorpusID:258291425

[25] Abdul Quamar, Vasilis Efthymiou, Chuan Lei, and Fatma Özcan. 2022. Natural Language Interfaces to Data. *Found. Trends Databases* 11, 4 (May 2022), 319–414.

[26] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *ArXiv* abs/1910.01108 (2019).

[27] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD - Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

[28] Harshit Varma, Abhijeet Awasthi, and Sunita Sarawagi. 2023. Conditional Tree Matching for Inference-Time Adaptation of Tree Prediction Models.

[29] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2024. MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL. arXiv:2312.11242 [cs.CL]

[30] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. 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*. 7567–7578.

[31] Bailin Wang, Wenpeng Yin, Xi Victoria Lin, and Caiming Xiong. 2021. Learning to Synthesize Data for Semantic Parsing. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2760–2766.

[32] Brandon T Willard and Rémi Louf. 2023. Efficient Guided Generation for LLMs. *arXiv preprint arXiv:2307.09702* (2023).

[33] Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. 2025. OpenSearch-SQL: Enhancing Text-to-SQL with Dynamic Few-shot and Consistency Alignment. *CoRR* abs/2502.14913 (2025). arXiv:2502.14913 doi:10.48550/ARXIV.2502.14913

[34] Zhewei Yao, Guoheng Sun, Lukasz Borchmann, Zheyu Shen, Minghang Deng, Bohan Zhai, Hao Zhang, Ang Li, and Yuxiong He. 2025. Arctic-Text2SQL-R1: Simple Rewards, Strong Reasoning in Text-to-SQL. arXiv:2505.20315 [cs.CL] https://arxiv.org/abs/2505.20315

[35] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. 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*. 3911–3921.

[36] 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. 2019. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. arXiv:1809.08887 [cs.CL]

[37] Bohan Zhai, Canwen Xu, Yuxiong He, and Zhewei Yao. 2025. ExCoT: Optimizing Reasoning for Text-to-SQL with Execution Feedback. arXiv:2503.19988 [cs.LG] https://arxiv.org/abs/2503.19988

[38] Hanchong Zhang, Ruisheng Cao, Lu Chen, Hongshen Xu, and Kai Yu. 2023. ACT-SQL: In-Context Learning for Text-to-SQL with Automatically-Generated Chain-of-Thought. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 3501–3532. doi:10.18653/v1/2023.findings-emnlp.227

[39] Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. 2025. Qwen3 Embedding: Advancing Text Embedding and Reranking Through Foundation Models. *arXiv preprint arXiv:2506.05176* (2025).

Received April 2025; revised July 2025; accepted August 2025