CHASE-SQL: Multi-Path Reasoning and Preference Optimized Candidate Selection in Text-to-SQL

Mohammadreza Pourreza[1*], Hailong Li[1*], Ruoxi Sun[1], Yeounoh Chung[1], Shayan Talaei[2], Gaurav Tarlok Kakkar[1], Yu Gan[1], Amin Saberi[2], Fatma Özcan[1], Sercan Ö. Arık[1]
[1]Google Cloud, Sunnyvale, CA, USA
[2]Stanford University, Stanford, CA, USA
{pourreza, hailongli, ruoxis, yeounoh}@google.com
{gkakkar, gany, fozcan, soarik}@google.com
{stalaei, saberi}@stanford.edu
*Equal contribution

October 4, 2024

Abstract

In tackling the challenges of large language model (LLM) performance for Text-to-SQL tasks, we introduce CHASE-SQL, a new framework that employs innovative strategies, using test-time compute in multi-agent modeling to improve candidate generation and selection. CHASE-SQL leverages LLMs' intrinsic knowledge to generate diverse and high-quality SQL candidates using different LLM generators with: (1) a divide-and-conquer method that decomposes complex queries into manageable sub-queries in a single LLM call; (2) chain-of-thought reasoning based on query execution plans, reflecting the steps a database engine takes during execution; and (3) a unique instance-aware synthetic example generation technique, which offers specific few-shot demonstrations tailored to test questions. To identify the best candidate, a selection agent is employed to rank the candidates through pairwise comparisons with a fine-tuned binary-candidates selection LLM. This selection approach has been demonstrated to be more robust over alternatives. The proposed generators-selector framework not only enhances the quality and diversity of SQL queries but also outperforms previous methods. Overall, our proposed CHASE-SQL achieves the state-of-the-art execution accuracy of 73.0 % and 73.01% on the test set and development set of the notable BIRD Text-to-SQL dataset benchmark, rendering CHASE-SQL the top submission of the leaderboard (at the time of paper submission).

1 Introduction

Text-to-SQL, as a bridge between human language and machine-readable structured query languages, is crucial for many use cases, converting natural language questions into executable SQL commands (Androutsopoulos et al., 1995; Li & Jagadish, 2014; Li et al., 2024c; Yu et al., 2018; **?**). By enabling users to interact with complex database systems without requiring SQL proficiency, Text-to-SQL empowers users to extract valuable insights, perform streamlined data exploration, make informed decisions, generate data-driven reports and mine better features for machine learning (Chen et al., 2023; Pérez-Mercado et al., 2023; Pourreza & Rafiei, 2024a; Pourreza et al., 2024; Sun et al., 2023; Wang et al., 2019; Xie et al., 2023). Furthermore, Text-to-SQL systems play a pivotal role in automating data analytics with complex reasoning and powering conversational agents, expanding their applications beyond traditional data retrieval (Sun et al., 2023; Xie et al., 2023). As data continues to grow exponentially, the ability to query databases efficiently without extensive SQL knowledge becomes increasingly vital for a broad range of applications.

Text-to-SQL can be considered a specialized form of code generation, with the contextual information potentially including the database schema, its metadata and along with the values. In the broader code generation domain, utilizing LLMs to generate a wide range of diverse candidates and select the best one has proven to be effective (Chen et al., 2021; Li et al., 2022; Ni et al., 2023). However, it is non-obvious what

leads to most effective candidate proposal and winner selector mechanisms. A straightforward yet effective approach involves generating candidates using zero-/few-shot or open-ended prompting, followed by selecting the best options utilizing self-consistency (Wang et al., 2022), which entails clustering candidates based on their execution outputs. This approach has demonstrated promising results in several studies (Lee et al., 2024; Maamari et al., 2024; Talaei et al., 2024; Wang et al., 2023). However, a single prompt design might not fully unleash the extensive Text-to-SQL knowledge of LLMs, and self-consistency methods might not be always effective. In fact, as illustrated in Table 1, the most consistent answers would not always be the correct ones, with an upper-bound performance 14% higher than that achieved through self-consistency. This substantial gap highlights the potential for significant improvement by implementing more effective selection methods to identify the best answer from the pool of candidate queries.

Building on the challenges outlined in the previous section, we propose novel approaches to improve LLM performance for Text-to-SQL by leveraging judiciously-designed test-time computations in an agentic framework. As indicated by the upper bound in Table 1, utilizing LLMs' intrinsic knowledge offers significant potential for improvement. We propose methods that generate a diverse set of high-quality candidate responses and apply a selection mechanism to identify the best answer. Achieving both high-quality and diverse candidate responses is critical for the success of scoring-based selection methods. Low diversity limits improvement potential and reduces the difference between self-consistency and scoring-based approaches. While techniques like increasing temperature or reordering prompt contents can boost diversity, they often compromise the quality of the candidates. To address this, we introduce effective candidate generators designed to enhance diversity while maintaining high-quality outputs. Specifically, we propose three distinct candidate generation approaches, each capable of producing high-quality responses. The first is inspired by the divide-and-conquer algorithm, which breaks down complex problems into smaller, manageable parts to handle difficult queries. The second employs a query execution-plan-based chain-of-thought strategy, where the reasoning process mirrors the steps a database engine takes during query execution. Lastly, we introduce a novel online synthetic example generation method, which helps the model better understand the underlying data schema of the test database. These methods, when used independently, can produce highly-accurate SQL outputs. To effectively select the best answer among candidates, we introduce a selection agent, trained with a classification objective, that assigns scores based on pairwise comparisons between candidate queries. With this agent, we construct a comparison matrix for all candidates and select the final response based on the highest cumulative score. By combining these candidate generation methods with the proposed scoring model, we create an ensemble approach that leverages the strengths of each strategy to significantly improve overall performance.

Table 1: Evaluating single-query generation vs. ensemble methods of self-consistency and the upper bound that can be achieved for Text-to-SQL with Gemini 1.5 Pro on the BIRD dev set. EX stands for execution accuracy.

Method	EX (%)
Single query	63.01
Self-consistency	68.84 (+ 5.84)
Upper-bound	**82.79** (+ 19.78)

We present comprehensive evaluations on the efficacy of proposed methodologies of CHASE-SQL. Our innovative candidate generation approaches demonstrate superior performance compared to traditional generic CoT prompts, illustrating their capability in guiding LLMs through the decomposition of complex problems into manageable intermediate steps. Furthermore, the proposed selection agent significantly outperforms conventional consistency-based methods, contributing to the state-of-the-art results. Specifically, CHASE-SQL reaches an execution accuracy of **73.01%** and **73.0%** on the development set and test set of the challenging BIRD Text-to-SQL dataset which outperforms all of the published and undisclosed methods on this benchmark, by a large margin.

2 Related Work

Early Text-to-SQL methods predominantly utilized sequence-to-sequence architectures, encoding user queries and database schemas using models such as Graph Neural Networks (GNNs), Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM) networks, and pre-trained transformer encoders (Cai et al., 2021; Cao et al., 2021; Hwang et al., 2019). On the decoding side, these systems employed either slot-filling or auto-regressive modelling approaches to construct the final SQL queries from the encoded inputs (Choi et al., 2021; Wang et al., 2019). Additionally, tabular language models like TaBERT (Yin et al., 2020),

TaPas (Herzig et al., 2020), and Grappa (Yu et al., 2020) have been developed to encode both tables and textual data effectively. However, the landscape has evolved with the widespread use of LLMs, which have largely replaced earlier methods with their superior performance (Katsogiannis-Meimarakis & Koutrika, 2023; Quamar et al., 2022). Initially, efforts concentrated on optimizing prompt designs for these LLMs (Dong et al., 2023; Gao et al., 2023; Pourreza & Rafiei, 2024a). Subsequent advancements have introduced more complex methodologies, including schema linking (Li et al., 2024b; Pourreza & Rafiei, 2024a,b; Talaei et al., 2024), self-correction or self-debugging (Chen et al., 2023; Talaei et al., 2024; Wang et al., 2023), and self-consistency techniques (Lee et al., 2024; Maamari et al., 2024; Sun et al., 2023; Talaei et al., 2024), further enhancing the performance by proposing complex LLM-based pipelines.

3 Methods

3.1 Overall Framework

This section outlines the proposed CHASE-SQL framework, which consists of four primary components: 1) Value retrieval, 2) Candidate generator, 3) Query fixer, and 4) Selection agent. As illustrated in Fig. 1. The proposed framework begins by retrieving relevant database values. Subsequently, all contextual information, including retrieved values, database metadata, and schema, is provided to an LLM to generate candidate queries. These candidate queries then undergo a fixing loop, and finally, all candidates are compared in a pairwise way using the trained selection agent to pick the correct answer. The following sections delve into the details of each component.

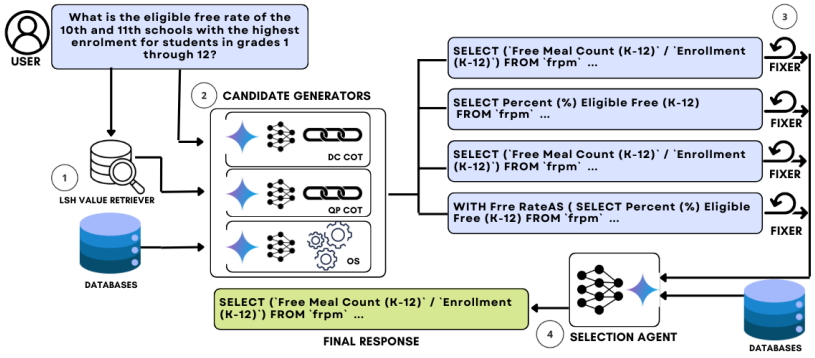

Figure 1: Overview of the proposed CHASE-SQL framework for Text-to-SQL, with value retrieval and using a selection agent for improve picking of the answers among the generated candidates along with a fixer to provide feedback for refinement of the outputs.

3.2 Value Retrieval

Databases might contain very high number of rows, with often only a few being relevant to a query. Retrieving relevant values is crucial as they can be used in various SQL clauses like 'WHERE' and 'HAVING'. Similar to the approach in (Talaei et al., 2024), we begin by extracting keywords from the given question using an LLM prompted with few-shot examples. For each keyword, we employ locality-sensitive hashing (LSH) (Datar et al., 2004) to retrieve the most syntactically-similar words, and re-rank them based on embedding-based similarity and edit distance. This approach is robust to typos in the question and considers keyword semantics during retrieval.

3.3 Multi-path Candidate Generation

As shown in Table 1, relying solely on consistency among responses can lead to sub-optimal performance. Therefore, we prioritize diversity in generation of multiple response candidates to increase the likelihood of generating at least one correct answer. Among the diverse responses generated by the candidate generators, we select one as the final response using a selection agent that compares candidates pairwise. To generate

diverse responses, we increase the next token sampling temperature, and also shuffle the order of columns and tables in the prompt.

Chain-of-Thought (CoT) prompting (Wei et al., 2022) has been proposed to enhance LLMs' reasoning abilities by conditioning their final responses on a step-by-step chain of reasoning. Most CoT prompting approaches rely on few-shot examples in the prompt to guide LLMs on thinking step-by-step, following the format $M = (q_i, r_i, s_i)$, where q_i is the example question, r_i is the reasoning path, and s_i is the ground truth SQL query for q_i. We employ two distinct reasoning methods and an online synthetic example generation approach. As shown in Fig. 3a, different generators can yield different outputs, indicating their effectiveness for specific questions and databases.

Divide and Conquer CoT: Divide-and-conquer perspective brings breaking down complex problems into smaller sub-problems, solving each individually, and then combining the solutions to obtain the final answer. Along these lines, we propose a CoT prompting approach that first decomposes the given question into smaller sub-problems using pseudo-SQL queries. In the 'conquer' step, the solutions to these sub-problems are aggregated to construct the final answer. Finally, an optimization step is applied to the constructed query to remove redundant clauses and conditions. This approach is particularly powerful handling complex scenarios that involve nested queries, e.g. intricate WHERE or HAVING conditions, and queries requiring advanced mathematical operations. In Appendix Fig. 17, we exemplify a question and its corresponding SQL query that was successfully solved using this generator, a scenario the other methods considered in this paper could not address due to the query's complex conditions and SQL clauses. For a more detailed view of the divide-and-conquer prompt, please see Appendix Fig. 16. Additionally, Alg. 1 outlines the step-by-step process of this strategy to generate the final SQL output using a single LLM call.

Algorithm 1 Divide and Conquer Chain-of-Thought (CoT) Strategy for Text-to-SQL.

Input: Set of human-annotated few-shot examples M, user question Q_u, target database D associated with the question, and a large language model (LLM) θ.

Divide:

$S_q \leftarrow \theta(M, D, Q_u)$ // *Decompose the original question Q_u into a set of sub-questions S_q*

$S_{sql} \leftarrow \emptyset$ // *Initialize an empty set S_{sql} to store partial SQL queries for each sub-question*

Conquer:

for each sub-question q_i in S_q **do**

  // *Generate a partial SQL query for each sub-question q_i*

  $S_{sql} \leftarrow S_{sql} \cup \{\theta(M, D, Q_u, q_1, ..., q_i, sql_1, ..., sql_{i-1})\}$

end for

Assemble:

$S_f \leftarrow \theta(M, D, Q_u, S_q, S_{sql})$ // *Assemble the final SQL query S_f from all sub-queries in S_{sql}*

return S_f

Query Plan CoT: A query (execution) plan is a sequence of steps that the database engine follows to access or modify the data described by a SQL command. When a SQL query is executed, the database management systems' query optimizers translate the SQL text into a query plan that the database engine can execute. This plan outlines how tables are accessed, how they are joined, and the specific operations performed on the data (see Appendix Fig. 19 as an example). Inspired by the step-by-step process that database engines use to execute SQL queries, we propose a reasoning strategy to construct the final SQL output. Query plans for any given SQL query can be obtained using the "EXPLAIN" command, which provides a detailed breakdown of execution steps. However, this output is often presented in a format that is difficult to interpret by LLMs (e.g. in SQLite). To address this, we convert the output of "EXPLAIN" command into a human-readable text format that aligns more closely with the pretraining data of LLMs. The human-readable version of query plans consists of three key steps: (1) identifying and locating the relevant tables for the question, (2) performing operations such as counting, filtering, or matching between tables, and (3) delivering the final result by selecting the appropriate columns to return. This reasoning method complements the divide-and-conquer CoT strategy. While the divide-and-conquer approach is better suited for decomposing complex questions, the query plan approach excels when questions require more reasoning over the relationships between different parts of the question and the database schema. It systematically explains which tables to scan, how to match columns, and how to apply filters. Appendix Fig. 20 shows an example of a question that was answered correctly only by this method. Appendix Fig. 18 provides the prompt used for this reasoning strategy.

Online Synthetic Example Generation: Using M demonstrations for few-shot in-context learning has shown promising results on various related tasks (Pourreza & Rafiei, 2024a). Besides helping with specifying the task and illustrate the step-by-step process deriving the output, demonstrations constructed using relevant tables and columns can also help the model understand the underlying data schema. Based on this insight, we propose a synthetic demonstration generation strategy for Text-to-SQL – given the user question Q_u, the target database D, and the selected columns t_i (using a column selection approach similar to (Talaei et al., 2024)).

Algorithm 2 Online Synthetic example generation strategy for Text-to-SQL.

Input: User question Q_u, additional user hint H_u, target database D and filtered relevant table columns t associated with the question, LLM θ, guidelines R_f for generating examples by SQL features, guidelines R_t for generating examples with filtered schema, and the numbers of examples to generate n_f, n_t respectively

$P \leftarrow \emptyset$ // $\{(q_i, s_i) \mid q_i, s_i \in \Sigma^*\}$, *where* q_i *is input question ,* s_i *is output SQL for the i-th example*

$P \leftarrow P \cup \{\theta(D, R_f, n_f)\}$ // *Generate n examples with entire database by common SQL features*

$P \leftarrow P \cup \{\theta(t, R_t, n_t)\}$ // *Generate examples with filtered columns to highlight correct schema usage*

return P

Algorithm 2 outlines the online synthetic example generation approach with two LLM generation steps. The first step focuses on generating illustrative examples with common SQL features described in the guideline R_f. The SQL features include equality and non-equality predicates, single table and multi-table JOIN, nested JOIN, ORDER BY and LIMIT, GROUP BY and HAVING, various aggregation functions. These are widely applicable SQL clauses and functions – the generated example SQL queries, incorporating these features, follow the BIRD SQL feature distribution (Appendix Fig 23a). The second step focuses on generating examples highlighting correct interpretation of the underlying data schema – the model θ is asked to generate examples using t_i and that are similar to the examples outlined in R_t. Appendix A.10 provides the prompts used for the example generation).

While a relevant example (e.g. showing a nested JOIN query with multiple tables) can be helpful for questions that require complex JOIN queries, it might also mislead the LLM for overuse (e.g. when a simple single table query is sufficient). This and the inherent ambiguity of natural language query q_i, for which we draw the examples by relevance, make the example selection challenging. Thus, we generate and inject the examples to the prompt online per q_i. We ask the LLM to generate many input-output pairs for in-context learning. The final set of synthetic examples for q_i contains examples generated with both R_f and R_t. This ensures that the example set is diverse both in SQL features/clauses and the choice of relevant tables/columns used. The diversity of the example set is desirable to avoid over-fitting the output to certain patterns (e.g., the model always writes a SQL with JOIN if shown mostly JOIN examples). Mixing various examples for various SQL features and database tables with and without column filtering is observed to result in better generation quality overall (please see Appendix Table 8).

3.4 Query Fixer

In some cases, LLMs might generate queries that are syntactically incorrect. These queries are clear candidates for correction, as they fail to provide the correct answers. To address this, we apply an LLM-based query fixer that leverages the self-reflection (Shinn et al., 2024) method. The fixer reflects on the previously generated query, using feedback such as syntax error details or empty result sets to guide the correction process. We continue this iterative fixing approach up to a specified number of attempts, β (set to three in this paper). Appendix Fig. 21 demonstrates the prompt used for this query fixing step.

3.5 Selection Agent

With three different methods for generating SQL queries, we can generate a set of candidate queries for any given question. The key challenge in this step is selecting the correct SQL query from this pool of candidates. A naive approach would be to measure consistency among the candidates by executing them, grouping them based on their execution results, and selecting a query from the largest group as the most likely correct answer. However, this would assume that the most consistent answer is always the best one, which is not always the case. Instead, we propose a more refined picking strategy, Algorithm 3, that relies on a selection agent. Given a set of candidates SQL queries $C = \{c_1, c_2, ..., c_n\}$, the final responses are selected by finding

the candidate that has the highest score assigned by the selection model. This model θ_p can take k candidates and rank them based on how accurately each of them answers the given question. Concretely, we formulate the selection of the final response as:

$$c_f = \arg\max_{c \in C} \left(\sum_{i=1}^{\binom{n}{k}} \theta_p(c_{i_1}, \ldots, c_{i_k} \mid Q_u, H_u, D) \right), \tag{1}$$

where Q_u refers to the user's question, H_u is the provided hint, and D is the target database from which the question is being asked. In Eq. 1, we pass k candidates to the selection model to be ranked, with k being between 1 and n. In the extreme case of $k = 1$, the model is unable to make comparisons between candidates, which complicates the evaluation process for the model. As k increases, comparing more candidates makes the process more challenging for the model, as it needs to consider different aspects simultaneously. Consequently, we set $k = 2$ and train a model with a classification objective to compare only two candidates at a time.

Having a set of high-quality and diverse candidates, the most straightforward solution is to employ off-the-shelf LLMs to make pairwise selections. However, experiments with Gemini-1.5-pro showed that using the LLM without fine-tuning resulted in only 58.01% binary classification accuracy. This is primarily due to the candidates being very similar to one another, requiring a fine-tuned model to learn the nuances and make more accurate decisions. To train the selection agent, we first generate candidate SQL queries on the training set (of Text-to-SQL benchmarks), and group them into clusters based on their execution results. For cases where at least one cluster contains correct queries and others contains incorrect ones, we create training examples in the form of tuples $(Q_u, C_i, C_j, D_{ij}, y_{ij})$, where Q_u is the user's question, C_i and C_j are the two candidate queries being compared, D_{ij} is the database schema used by both candidates, and $y_{ij} \in 0, 1$ is the label indicating whether C_i or C_j is the correct query. To avoid order bias during training, we randomly shuffle the order of correct and incorrect queries in each pair. Since the number of cases with both correct and incorrect candidates is limited, for instances where no correct candidate exists, we include the ground truth SQL query in the prompt as a hint to guide the model in generating correct candidates.

Algorithm 3 Picking the final SQL query from a pool of candidates.

Input: Set of candidate SQL queries $C = \{c_1, c_2, ..., c_n\}$, user question Q_u, hint H_u, target database D, and a selection model θ_p, $er(c_i, D)$ as the execution result of c_i on D

$r_i \leftarrow 0$ for all $c_i \in C$ // *Initialize the score r_i for each candidate query to zero*
for each distinct pair (c_i, c_j) where $i \neq j$ **do**
  if $er(c_i, D) = er(c_j, D)$ **then**
    $w \leftarrow i$ // *c_i is the winner if the execution results match*
  else
    $S_{i,j} \leftarrow schema\_union(c_i, c_j, D)$ // *Construct union of schemas used in c_i and c_j*
    $w \leftarrow \theta_p(S_{i,j}, Q_u, H_u, c_i, c_j) w \in \{i, j\}$ // *Use binary classifier θ_p to select the winner, $w \in \{i, j\}$*
  end if
  $r_w \leftarrow r_w + 1$ // *Increase the score of the winner c_w by 1*
end for
$c_f \leftarrow \arg\max_{c_i \in C} r_i$ // *Select the candidate with the highest score as the final SQL query c_f*
return c_f

In the pseudo-code for Algorithm 3, we begin by initializing a score of zero for each candidate query. Then, for every distinct pair of queries (c_i, c_j), we compare both (c_i, c_j) and (c_j, c_i) to mitigate any order bias, ensuring that both candidates in a pair are fairly evaluated. If both queries produce the same execution result on the database, we mark one as the winner and increment its score, as these results suggest consistency. If the execution results differ, we generate a union of the schema used by both queries and use the binary classifier to determine which query is more likely to be correct. The classifier takes into account the question, the two candidate queries, and the combined schema to make its decision. The winner's score is then updated accordingly. After all comparisons, the candidate with the highest score is selected as the final query. In the rare case of a tie in the final scores, we break the tie by selecting one of the candidates arbitrarily.