

MageSQL: Enhancing In-context Learning for Text-to-SQL Applications with Large Language Models

Chen Shen
Megagon Labs
Mountain View, United States
chen_s@megagon.ai

Sajjadur Rahman*
Adobe
San Jose, United States
sajjadurr@adobe.com

Jin Wang
Megagon Labs
Mountain View, United States
jin@megagon.ai

Eser Kandogan
Megagon Labs
Mountain View, United States
eser@megagon.ai

ABSTRACT

The text-to-SQL problem aims to translate natural language questions into SQL statements to ease the interaction between database systems and end users. Recently, Large Language Models (LLMs) have exhibited impressive capabilities in a variety of tasks, including text-to-SQL. While prior works have explored various strategies for prompting LLMs to generate SQL statements, they still fall short of fully harnessing the power of LLM due to the lack of (1) high-quality contextual information when constructing the prompts and (2) robust feedback mechanisms to correct translation errors. To address these challenges, we propose MageSQL, a text-to-SQL approach based on in-context learning over LLMs. MageSQL explores a suite of techniques that leverage the syntax and semantics of SQL queries to identify relevant few-shot demonstrations as context for prompting LLMs. In particular, we introduce a graph-based demonstration selection method – the first of its kind in the text-to-SQL problem – that leverages graph contrastive learning adapted with SQL-specific data augmentation strategies. Furthermore, an error correction module is proposed to detect and fix potential inaccuracies in the generated SQL query. We conduct comprehensive evaluations on several benchmarking datasets. The results show that our proposed methods outperform state-of-the-art methods by an obvious margin.

VLDB Workshop Reference Format:

Chen Shen, Jin Wang, Sajjadur Rahman*, and Eser Kandogan. MageSQL: Enhancing In-context Learning for Text-to-SQL Applications with Large Language Models. VLDB 2024 Workshop: Applied AI for Database Systems and Applications (AIDB 2025).

VLDB Workshop Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/megagonlabs/magesql-full>.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment. Proceedings of the VLDB Endowment. ISSN 2150-8097.

1 INTRODUCTION

Given a relational database, the text-to-SQL problem automatically translates the natural language question into an SQL statement that queries the database system to find the results. This problem is increasingly critical for improving the accessibility and usability of relational database systems for a broad range of users, especially non-technical users [11, 13, 16] who are not familiar with database concepts and SQL.

There is a long stream of research on the topic of text-to-SQL from both database and NLP communities. Earlier studies [16, 36] employed rule-based methods that first converted the natural language question into an intermediate representation and then mapped them into SQL abstract syntax trees with heuristic rules. Later, techniques emerged that utilized deep learning techniques to develop solutions [12, 22, 32, 38], which can support cross-domain adaption as well as handle complex queries. The basic idea is to formulate text-to-SQL as a machine translation problem and then utilize different variants of models with encoder-decoder architecture to solve it. Follow-up work such as [7, 11, 20] further proposed sketching-based solutions to regularize the syntax of generated SQL queries via pre-defined templates.

Most recently, advances in the era of Large Language Models (LLMs) have brought new opportunities to the problem of text-to-SQL. Pre-trained LLMs such as GPT-4 [30], LLaMA [42] and Codex [6] have shown superior abilities in understanding human instructions as well as generating structured output. Such LLMs are generative models that take a sequence of tokens as input and generate a sequence of tokens as output. Various studies have shown that input to the models, i.e., prompts, is critical in achieving desired results. As such, *prompt engineering* has become an important methodology in utilizing LLMs [26]. Consequently, departing from previous algorithmic approaches, the LLM-based solutions focused on engineering effective prompt strategies to improve the overall performance [13, 29, 31].

However, some research challenges remain unresolved regarding fully utilizing the impressive capabilities of LLMs. Firstly, previous results show that while LLM-based solutions are good at understanding natural language questions, there are still various issues in generating SQL statements. This is mainly due to the lack of effective examples in the prompt that guide the LLM for SQL generation.

*Work done while at Megagon Labs.

Secondly, while previous LLM-based solutions focused on developing advanced reasoning techniques, e.g. chain-of-thought [46], to facilitate the generation process, they treated the output of LLMs as the final results to be executed in the database systems. However, since it is well known that the output of LLMs might have uncertainties such as hallucinations, such a practice in previous studies might fail to address the potential issues in the output. Let’s illustrate these through a few examples in Figure 1:

EXAMPLE 1. As shown in Figure 1, the top example illustrates a case with two-shot learning, where the ground truth requires the usage of conjunctions, yet the demonstration examples did not include any. Consequently, without proper guidance, an LLM may struggle to detect the need for a conjunction, leading to potential mistakes. The bottom example shows a case where the prediction from LLM is in fact very close to the ground truth. However, there is a slight mismatch between the LLM output and ground truth, which could be easily fixed by rule-based format adjustment. These examples clearly illustrate the need for proper guidance of LLM generation through better demonstration examples and post-processing to resolve final issues, even when the LLM successfully understands the query intent.

| | |
|----------------|---|
| Question | Show the document id with paragraph text 'Brazil' and 'Ireland'. |
| Demonstrations | <p>### Answer the following question: What is the name of the conductor who has worked the greatest number of years? SELECT Name FROM conductor ORDER BY Year_of_Work DESC LIMIT 1</p> <p>### Answer the following question: What is the maximum capacity and the average of all stadiums? select max(capacity), average from stadium</p> |
| Predicted SQL | SELECT Document_ID FROM Paragraphs WHERE Paragraph_Text IN ('Brazil', 'Ireland') |
| Gold SQL | SELECT document_id FROM Paragraphs WHERE paragraph_text = 'Brazil' INTERSECT SELECT document_id FROM Paragraphs WHERE paragraph_text = 'Ireland' |

| | |
|---------------|--|
| Question | What are the descriptions for all the math courses? |
| Predicted SQL | SELECT course_description FROM Courses WHERE course_name LIKE "%math%" |
| Gold SQL | SELECT course_description FROM Courses WHERE course_name = 'math' |

Figure 1: Motivation Examples

In this paper, we propose MageSQL, a new framework for Text-to-SQL based on in-context learning over LLMs. First, our key observation is that high-quality demonstration examples in few-shot learning is important to improving effectiveness. Toward this goal, it is critical that the examples are similar to the questions and potential answers to provide meaningful guidance to SQL generation. We explore strategies motivated by previous work and recognize that it is crucial to consider both structural and semantic information in the selection to ensure that better examples with a similar structure to the ground-truth SQL are included. To this end, we first develop a structure-based solution that uses the similarity between Abstract

Syntax Trees of SQL statements as the metric for demonstration selection. Next, we develop a graph embedding-based solution that can capture both structural similarity and semantics of SQL statements. This could be realized by constructing a graph for each SQL, which consists of both the syntactic parsing results of SQL and the schema of tables associated with it. Then the similarity between SQL statements could be evaluated via that between their graph embeddings. To reach this goal, we propose a graph contrastive learning [14, 41, 49, 55] based framework to learn the graph encoder for node embedding in a fully unsupervised manner. Therefore, the framework is generalizable to any domain and easier to adopt as no human supervision via annotation is required. Then the embedding of an SQL, i.e., the whole graph, could be obtained by aggregating the embedding of all its nodes.

In addition, as part of post-processing, we develop a new error correction module to fix the potential errors in the output. Specifically, we employ two categories of error-correction strategies: rule-based and prompt-based. The rule-based strategy aims to correct minor syntax and string format errors and, thus, is very lightweight. On the other hand, the prompt-based strategy would do another round of prompts by asking an LLM to rewrite the generated SQL using a set of predefined guidelines so as to correct the errors. The goal of this approach is to resolve more complicated errors that cannot be easily handled by hand-crafted rules. This is inspired by recent work [37, 47] of *multi-agent system*, where complex tasks are finished through a collaboration of multiple agents, making our efforts in each step suitable for deployment in such systems as independent agents [39].

The contribution of this paper is summarized as follows:

- We propose a new framework to enhance in-context learning for the Text-to-SQL task based on LLMs.
- We investigate the strategies of demonstration selection under the few-shot setting and employ structure similarity to find high-quality examples.
- We introduce the first-of-its-kind graph embedding-based solution for demonstration selection in text-to-SQL, which resulted in up to 5.4% performance gain over previous selection methods.
- We developed a novel error correction module to fix the potential errors in the generated SQL to improve the overall performance.
- We conducted extensive experiments on two popular text-to-SQL benchmarks by up to 13.2% in Execution Accuracy compared to previous LLM-based solutions. The results showed that our proposed method outperformed state-of-the-art methods.

The rest of the paper is organized as follows: Section 2 provides essential background about LLM and the text-to-SQL problem. Section 3 presents our proposed framework. Section 4 introduces the evaluation results. Section 5 surveys the related works. Section 6 concludes the whole paper.

2 PRELIMINARY

2.1 Large Language Model Terminologies

Recent years have witnessed a rapid advance in the application of large-scale, pre-trained language models in almost all NLP tasks.

The milestone work of pre-trained Language Model (PLM) is BERT [9] that aimed at learning contextual word embeddings by pre-training a bi-directional transformer-based architecture, comprising a stack of self-attention layers that calculates distributed representations based on the similarity against all tokens and produces contextual embeddings for each input token. There are two steps in the development of PLMs: pre-training and fine-tuning. In the pre-training step, the language model is trained on a large unlabeled corpus such as Wikipedia to gain deep language understanding via the pre-training tasks. The pre-trained model could be further fine-tuned for specific target downstream tasks with labeled training data.

Large Language Models (LLMs) have emerged as a new paradigm of research works in a variety of research fields. Many pre-trained LLMs have been released to provide public APIs or checkpoints, such as GPT [30], LLaMA [42], Palm [8] and CodeX [6]. Compared with PLMs, LLMs are pre-trained following similar methodologies but have a much larger number of parameters. For example, the number of parameters of pre-trained BERT and GPT-3 is 340 million and 178 billion, respectively. Due to their huge size, a common way to utilize LLM without incurring significant overhead is to provide text instructions to guide generation, known as prompt engineering [26]. LLMs have demonstrated remarkable in-context learning abilities [4], guiding predictions based on a relatively few pieces as additional input. Generally speaking, there are two different settings of in-context learning: (1) *few-shot learning*, when demonstration examples are included in the prompt as input; (2) *zero-shot learning*, when no demonstration is presented.

2.2 The Text to SQL Problem

| | |
|-------------|--|
| Instruction | ### Complete sqlite SQL query only and with no explanation. |
| Context | <p>### Some example pairs of question and corresponding SQL query are provided based on similar problems:</p> <p>### Answer the following question: What is the nationality of " Kevin Spacey " ?</p> <p>SELECT nationality FROM director WHERE name = "director_name0";</p> <p>### Answer the following question: What is the number of movies produced in 2013 ?</p> <p>SELECT COUNT (DISTINCT title) FROM movie WHERE release_year = 2013;</p> <p>### Given the following database schema:</p> <p>CREATE TABLE "singer" ("Singer_ID" int,"Name" text,"Birth_Year" real,"Net_Worth_Millions" real,"Citizenship" text,PRIMARY KEY ("Singer_ID"))</p> <p>CREATE TABLE "song" ("Song_ID" int,"Title" text,"Singer_ID" int,"Sales" real,"Highest_Position" real,PRIMARY KEY ("Song_ID"),FOREIGN KEY ("Singer_ID") REFERENCES `singer` ("Singer_ID"))</p> |
| Question | ### Answer the following question: What is the count of singers? SELECT |

Figure 2: An Example of the Prompt Template

Given a natural language question Q and a database D , the text-to-SQL problem aims to find an SQL query Y that corresponds to the question. The LLM-based solutions for text-to-SQL [10, 13, 29, 31] formulate it as a generation problem that employs a prompt P for the LLM M . It estimates the probability distribution over the potential SQL queries Y and generates the SQL statement token-by-token.

This generation process could be formulated as Equation 1:

$$Pr_M(Y|P, D, Q) = \prod_{i=1}^{|Y|} Pr_M(Y_i|P, D, Q, Y[0...i-1]) \quad (1)$$

where $Y[0...i-1]$ is the sequence generated by the model so far before step i .

An example of a prompt template over LLM for the Text-to-SQL problem is shown in Figure 2, where the SQL generated by GPT-4 is: `SELECT count(*) FROM singer`. The prompt for text-to-SQL typically includes three key components:

- **Instruction**, giving the general task descriptions.
- **Context**, providing the necessary context for task and demonstration examples. This is the most important component of a prompt.
- **Question**, describing the expected answer from (e.g. natural language question)

In this example, the prompt’s context includes two parts: (1) *Demonstration*, which provides some examples for few-shot learning, and (2) *Schema*, which displays the schema information of the targeted database to offer hints for generating the SQL. As such, an essential goal is to provide high-quality context with an appropriate strategy to construct the prompt.

3 METHODOLOGY

3.1 Overview

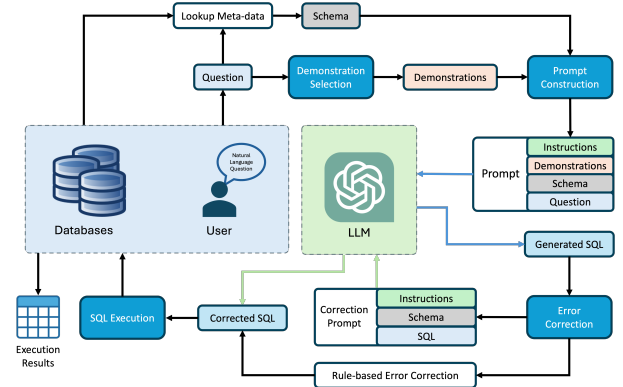


Figure 3: Overall Framework.

The overall framework of MageSQL is shown in Figure 3. Given a question in natural language, we use the question to fetch (1) database schema and (2) demonstration examples to construct the prompt. The database schema suggests the necessary background specifics for SQL statements, such as the names and types of tables and attributes. Following the previous work [13], we use the corresponding Data Definition Language (DDL) to describe the schema. The demonstration examples provide useful contextual information to the LLM to facilitate the generation of SQL. Following the discussions in previous works, we explored several options and proposed two new structure-based methods in Section 3.2 and Section 3.3, respectively. After obtaining the initial output SQL from LLM, we further perform error correction in the post-processing step to find and fix potential errors in Section 3.4.

3.2 Demonstration Selection Strategies: Basics

In the process of prompt engineering over LLM, one essential way is through in-context learning [4] where LLM could use the contextual examples and condition its generation by recognizing patterns in the input. This allows LLMs to perform new tasks during inference without any task-specific fine-tuning. Previous studies [25, 28] have shown that it is essential to select helpful examples to support the in-context learning with few-shot examples over LLM in different kinds of applications. Similarly, it is also essential for prompt construction in the Text-to-SQL problem.

To this end, we focused on developing effective techniques to select demonstrations in the prompt construction process for the Text-to-SQL problem. According to previous studies [13, 29], it is essential to select examples that are similar to the given question instance. Following this route, we address this problem by first defining a similarity metric to evaluate the relevance between a given instance and candidate examples and then selecting the top-ranked ones. The candidates of demonstration examples could be pairs of a natural language question and the corresponding SQL that do not appear in the set of questions (e.g. dev and test sets in a benchmarking dataset). We will start from 3 basic approaches to select k demonstration examples motivated by the high level idea in previous studies [13, 29]:

Random. In this approach, k examples are selected via random sampling from the available candidates. It is considered the baseline method in several previous studies.

Hardness In this approach, we use query difficulty as a measure. Examples are randomly selected from the group of instances that has the same level of hardness with question instance. The Spider dataset [50] provides a tag of difficulty level (easy, medium, hard, extra) for each instance, and thus we can directly use it for selection. For other datasets without such information in the metadata, we can use some rule-based heuristics to decide the hardness following the practice of previous works [50, 53].

Question Similarity This approach uses the string similarity between questions as the measure. Here, we choose Jaccard Similarity as the metric and select results with top- k highest scores as the results. Unlike the other two methods, the results of this method are deterministic since it does not involve random selection.

Although existing works have explored various strategies for selecting demonstrations to be included as few-shot examples, they have certain deficiencies in the domain of SQL (will be illustrated in our empirical observations in Section 4.3). Since an LLM generates the SQL based on the input prompt, it is essential to provide some examples with SQL that is similar to the expected output. We developed a new demonstration selection strategy based on *structure similarity* of SQL statements to address this issue. To describe the structure of an SQL, we consider its Abstract Syntax Tree (AST), which consists of the relational operators. AST is a general data structure that is independent of the specific database systems. It is also a relatively lightweight data structure and could be generated without an underlying database system. Specifically, we generate the trees through the third-party parsing tool sqlglot¹. In this tree-based solution, we focus on the structure information

and ignore the exact names of tables, attributes, and predicates. Only the type information of tree nodes is kept as the *node label*. Examples of node label include SELECT, WHERE, TABLE, aggregation operations (e.g. MIN, MAX, GROUP BY), conjunctions (e.g. INTERSECTION, EXCEPT, UNION), HAVING, and ORDER BY etc.

After transforming the SQL into the above tree structure, we then evaluate the structure similarity. Here, we choose *tree edit distance* [2] as the similarity metric. Basically, given two labeled trees, tree edit distance is the minimum number of edit operations that is needed to transform one tree into another. There are three kinds of edit operations:

- Insertion: insert a node between an existing node and a subsequence of consecutive children of this node;
- Deletion: delete a node and connect its children to its parent, maintaining the order;
- Substitution: rename the label of a node.

We will select examples with the top- k smallest tree edit distance from the question instance as the demonstration.

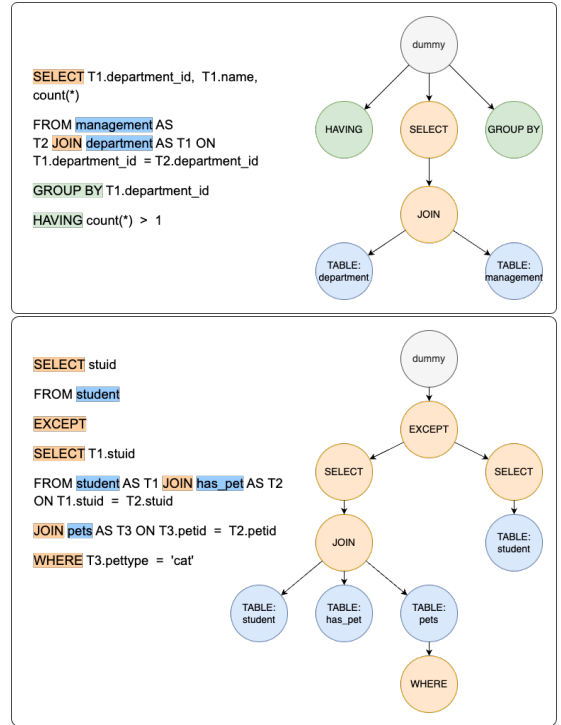


Figure 4: Examples of Tree Edit Distance between ASTs of SQL. Names of tables are kept just for reference but are not considered as node labels in measurement

EXAMPLE 2. An example of using tree edit distance to evaluate the structure similarity between SQL clauses is shown in Figure 4. We transformed the two SQL clauses into the AST and used different colors to denote different node labels. To transform the AST of the first SQL into that of the second one, we need the following operations: 3 deletions on the HAVING, GROUP BY, and TABLE: department nodes, 1 substitution to replace the SELECT with EXCEPT, 6 insertions to insert

¹<https://github.com/tobymao/sqlglot>

the left subtree of the EXCEPT node. As such, the tree edit distance is 10 in total. In this example, these two instances are not similar and should not be considered as a demonstration example for each other.

Although tree edit distance could accurately reflect the structure similarity between SQL clauses, its computational cost is $O(n^3)$, where n is the number of nodes in the tree. Since the candidate set of examples could be very large, computing the tree edit distance between the given instance and all potential candidate examples is very expensive. To address this problem, we adopt the idea of pq-gram from a previous study [1] to compute an estimation instead of the exact value of the tree edit distance. It is a signature that can help estimate the tree edit distance between two trees with less cost. To this end, we must obtain a set of pq-grams for each tree. Suppose the set of pq-grams for two trees is L_1 and L_2 , respectively; the pq-gram distance between the two trees can be calculated as $|L_1 \cup L_2| - 2 * |L_1 \cap L_2|$. The pq-gram distance could be calculated in $O(n \log n)$ time and serves as a lower bound of tree edit distance. And we will use the pq-gram distance between two ASTs instead of actual tree edit distance to select the demonstration examples. Due to the space limitation, here we omit the details of computation and proof of correctness, which could be found in [1].

One remaining issue to be resolved is that when constructing the prompt for a question, we only have the natural language question but not the actual SQL. Our solution is to first conduct a prompt with zero-shot learning to generate an initial SQL and use it as the query to find structurally similar examples. The extra overhead would be trivial since the prompt for zero-shot learning is much shorter than that with demonstration examples.

3.3 Graph based Demonstration Selection

While the above tree-based solution could capture the structural information of SQL statements, it still did not consider some important information, such as predicate values and column names in the involved tables. In addition, the structural similarity is evaluated by tree edit distance, which is based on syntactic similarity and thus might lose some latent structural information. In this section, we propose a graph-based demonstration selection approach to address such issues and further improve performance. Compared with the tree structure, the graph can carry not only richer structural information but also additional semantics. The basic idea is to construct a directed acyclic graph (DAG) to represent each SQL statement. In this way, the similarity between two SQL statements could be evaluated by that between their corresponding DAGs.

To reach this goal, the first step is to construct the graph (DAG) from a SQL statement. We extend the Abstract Syntax Tree (AST) representation described above by incorporating additional information. The graph consists of five types of node labels: (dummy) Root, SQL Keyword, Table, Column, and Value. Each unique SQL keyword is represented as an individual node (e.g., multiple JOIN nodes for a query). In contrast, identical table or column names are merged into a single node to maintain subgraph connectivity. Beyond the basic AST structure illustrated in Figure 4, we also include table and column names, as well as predicate values in the SQL query. After defining the nodes, we then add edges to capture relationships between them based on the following rules: (1) between each operator and the associated table or column name, (2)

between each column and the table it belongs to, and (3) between a predicate value and its corresponding literal operator. Additionally, SQL keywords like HAVING and GROUP BY are connected to their parent SELECT nodes. An example of representing SQL with the DAG is shown in Figure 5. Here, we first obtain the tree structure, which consists of essential operators in the SQL query. Next, we further parse the predicates and identify the columns (yellow nodes), literals (e.g. EQ), and values (green nodes). Finally, we add edges between such newly created nodes and the existing nodes corresponding to operators and obtain the graph.

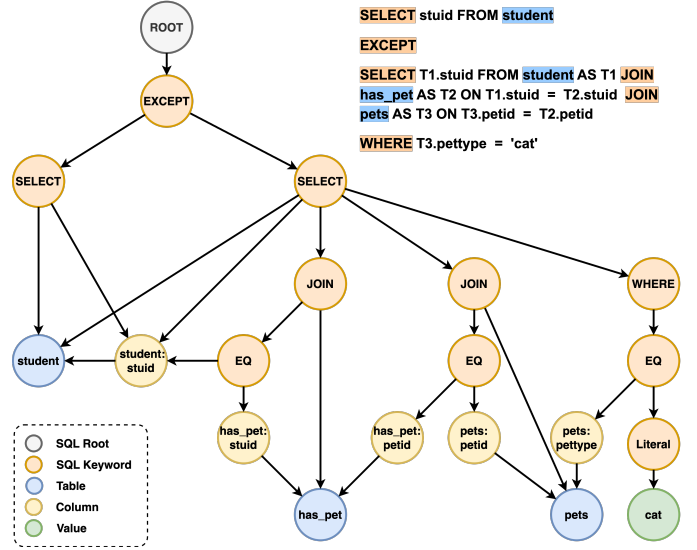


Figure 5: An Example of a SQL Statement and its Graph Representation.

Given such graph structures, we could evaluate the similarity based on cosine similarity between graph embeddings. To this end, we need to train a node encoder for the SQL graph. However, there is no labeled training instance, and the training process needs to be conducted in a fully unsupervised manner. To satisfy such needs, we employ the technique of graph contrastive learning [41, 49] as the solution. Graph contrastive learning is a variant of self-supervised learning that enables the training of graph encoders, such as Graph Neural Networks (GNNs), without human annotations. This is realized by constructing multiple graph views via stochastic augmentation of the input graph and then learning representations by contrasting positive samples against negative ones [55]. As illustrated in previous studies, two important factors of graph contrastive learning are contrastive instances and contrastive objective. While we can continue to employ the loss function from the previous studies as the objective, we need to consider the semantics of SQL when defining the graph augmentation operations for creating contrastive instances. According to our definition of the SQL graph, if we directly apply operations from previous studies [41, 49], we might end up with a result that is corresponding to a totally different SQL statement or even an invalid one. For example, if the node corresponding to a JOIN operation is replaced with a SELECT one, although the topological structure of

a SQL might still be close to the original instance, the structure of the corresponding SQL query will change greatly.

To keep the basic semantics of SQL statements when creating the contrastive instances, we define the following operations to perform augmentation of the input graph.

Feature Masking This operator randomly masks the node feature with a <MASK> token in LLM; while nodes with essential keywords (ROOT, SELECT, JOIN, WHERE, GROUP, ORDER) will not be masked.

Keyword Replacement This operator selects the SQL keywords that can be replaced while still keeping a valid SQL. These SQL keywords include logical and comparison operators (e.g., EQ, AND), arithmetic operators (e.g., ADD, DIV), and aggregations (e.g., COUNT, SUM, MIN). Each selected keyword node will be randomly replaced with a valid keyword node of the same type. For example, GT (>) could be replaced by LT (<), GTE (≥) and LTE (≤).

Value Replacement This operator selects nodes with type "VALUE", then replace it with new random values having the same data type (BOOLEAN, INT, FLOAT, STRING). Especially, for values that refer to partial match in SQL (e.g., "%USA"), only the partial string will be replaced (e.g., "%USA" to "%Canada").

Database Replacement This operator replaces the whole tables and columns that belong to one database with tables and columns in another database. It prefers to select new columns that have the same column type (e.g., numerical) as the original one to ensure that the augmented graph is a valid SQL.

Predicate Modification This operator chooses the predicate (e.g., WHERE, HAVING clause) of a SQL statement and then either randomly drops either the entire predicate or simplifies the condition in the predicate (e.g. "WHERE A=1 AND B=2" to "WHERE B=2").

Join Simplification For SELECT nodes with more than one JOIN node as neighbors, this operator randomly drops one JOIN node and corresponding clause. If there are nodes (e.g., TABLE or COLUMN) in such clause that are also connected to other nodes associated with essential SQL keywords, they will be kept.

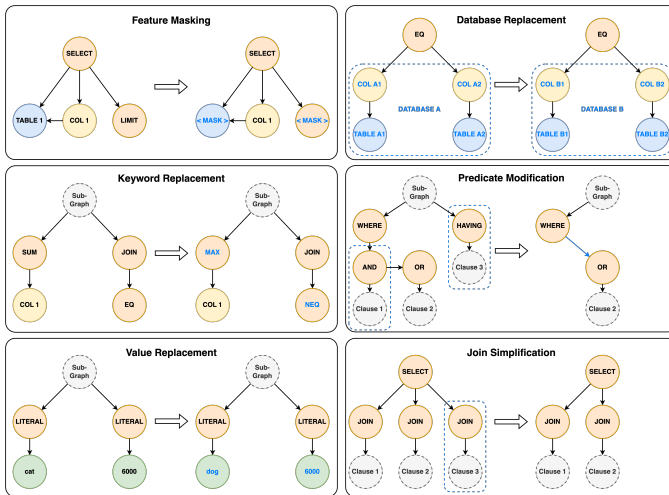


Figure 6: Examples for Graph Augmentation Operators. The blue font highlights the operations

EXAMPLE 3. We provide examples of the above-defined graph operations in Figure 6. For the SQL graph in Figure 5, the Feature Masking operator might randomly mask column node student:stuid and EQ keyword to <MASK> tokens. Keyword Replacement operator might replace the EQ under second JOIN with NEQ. Value Replacement operator might replace value node cat with bird. Database Replacement operator might replace table nodes student, has_pet, pets with singer, singer_in_concert, concert, respectively, and replace column nodes student:stuid, has_pet:stuid, has_pet:petid, pets:petid and pets:pettype with singer:Singer_ID, singer_in_concert:Singer_ID, singer_in_concert:concert_ID, concert:concert_ID and concert:Theme, respectively. Predicate Modification operator might drop WHERE node and its successors (pets will not be dropped because other parts also use it in the graph). The JOIN Simplification operator might drop the second JOIN node and its successors.

With the help of such operators, we are then able to obtain the contrastive instances. Given an original instance, we will randomly apply an operation defined above over it to obtain a positive instance; Meanwhile, the negative instances could be obtained by randomly sampling from the rest of the instances.

EXAMPLE 4. We provide an example of the above approaches of generating positive and negative instances for graph contrastive learning. Given the DAG for SQL shown in Fig 5, one positive instance could be obtained by applying the Predicate Modification operator to drop the where clause. Consequently, the corresponding SQL statement is "SELECT stuid FROM student EXCEPT SELECT T1.stuid FROM student AS T1 JOIN has_pet AS T2 ON T1.stuid = T2.stuid JOIN pets AS T3 ON T3.petid = T2.petid", which shares a similar structure with the original one. Meanwhile, a negative instance could be randomly selected from the rest of the dataset, an example could be the DAG corresponding to the SQL "SELECT Theme FROM farm_competition ORDER BY YEAR ASC".

Next, we employ the instances created with above approaches to train a graph encoder. In this part, the graph encoder can be implemented using various Graph Neural Network (GNN) architectures. In our implementation, we utilize a 2-layer Graph Attention Network (GAT) [44] as the encoder. We denote a graph as $G = (V, E, L)$, where $V = \{v_i\}_{i=1}^n$ and $E \subseteq V \times V$ denotes the set of nodes and edges, respectively. And L denotes the labeling function that assigns a label to each node. Each node v is initialized with concatenated features combining one-hot encoding of its node label \mathbf{l}_v and text embedding \mathbf{e}_v as illustrated in Equation 2.

$$\mathbf{h}_v^{(0)} = \text{CONCAT}(\mathbf{l}_v, \mathbf{e}_v) \quad (2)$$

Here, we obtained the text embedding by encoding the texts associated with a node, e.g., SQL keyword, column name, value, etc, with the pre-trained SentenceBert [34] model. With this node representation, we compute the propagation of representation at GNN layer k as Equation 3:

$$\mathbf{h}_v^{(k+1)} = \text{AGG} \left(\left\{ \text{COMBINE}(\mathbf{h}_v^{(k)}, \mathbf{h}_u^{(k)}) \mid u \in \mathcal{N}(v) \right\} \right) \quad (3)$$

where $\mathbf{h}_v^{(k)}$ is the node representation at layer k ; $\text{AGG}_{(k)}(\cdot)$ is the aggregation function that aggregates the information from neighbor nodes during the message-passing process; $\text{COMBINE}_{(k)}(\cdot)$ is the

function to merge node features with features aggregated from neighbors in the GNN layer; and $\mathcal{N}(v)$ denotes the set of neighbors of node v in the graph.

After obtaining each node embedding in the above method, we use graph readout function [41] to generate the graph embedding \mathbf{h}_G by aggregating that of all nodes as shown in Equation 4:

$$\mathbf{h}_G = \text{READOUT} \left(\{ \mathbf{h}_v^{(n)} \mid v \in \mathcal{V} \} \right) \quad (4)$$

where the READOUT layer combines mean, sum, and max aggregations over node embeddings in our implementation.

Then, the final graph embedding is obtained by adding a two-layer MLP projection head on top of the aggregated node embeddings as illustrated in Equation 5:

$$\mathbf{z}_G = \text{MLP}(\mathbf{h}_G) \quad (5)$$

The contrastive loss maximizes the similarity between an anchor graph and its positive samples while minimizing similarity with negative samples. We used normalized temperature-scaled cross-entropy loss NT-Xent widely utilized in previous studies [40, 43]. The details are shown in Equation 6:

$$\mathcal{L}_i = -\log \frac{\sum_{j=1}^{n_{\text{positive}}} \exp \left(\frac{\text{sim}(\mathbf{z}_i, \mathbf{z}_j^+)}{\tau} \right)}{\sum_{j=1}^{n_{\text{positive}}} \exp \left(\frac{\text{sim}(\mathbf{z}_i, \mathbf{z}_j^+)}{\tau} \right) + \sum_{k=1}^{n_{\text{negative}}} \exp \left(\frac{\text{sim}(\mathbf{z}_i, \mathbf{z}_k^-)}{\tau} \right)} \quad (6)$$

where \mathbf{z}_i , \mathbf{z}_j^+ and \mathbf{z}_k^- is the embedding of the anchor graph, the positive graph for the anchor and negative graph for the anchor, respectively; n_{positive} and n_{negative} is the number of positive graphs and negative graphs per anchor, respectively; $\text{sim}(\cdot)$ is the cosine similarity between two embeddings; τ is the temperature parameter for scaling the similarity scores.

3.4 Error Correction

Although LLMs are powerful in generating SQL statements based on input questions, some output statements might still be invalid due to a large training corpus beyond SQL and data that is not strictly compliant with SQL syntax. Potential errors also exist due to a lack of understanding of the contextual information or the question in the prompt. To address such issues, we propose an error correction module that automatically fixes such errors in the post-processing step. We proposed two kinds of error correction methods: rule-based and prompt-based.

First of all, we develop a set of rules to fix some simple errors based on the efforts of analyzing typical mistakes, an incomplete list of examples is as follows

String Format. Sometimes, the structure of generated SQL aligns with the ground truth, but there are mismatches between the values in the predicates, resulting in different execution results with the golden SQL. If such a mismatch is caused by string format issues such as spelling and cases, we can fix it via rules that align the values in the generated SQL with those in the database.

Mismatch in Schema. The LLM might involve non-existent or incorrect names of tables and attributes in the output due to hallucination. We will look up the metadata to ensure all the table and attribute names exist. If we find non-existing ones, we replace

them with the most similar ones from the metadata to ensure the generated SQL is valid.

Invalid Aggregation. We will fix the invalid aggregations, such as MIN and MAX, over non-numerical attributes or COUNT on multiple attributes. For the former case, we will directly remove it from the generated SQL; For the latter case, we will replace the attribute with the first attribute or *.

Join Condition. If the join condition happens between keys that are not joinable, we will replace it with foreign keys that are joinable between the two tables. If that does not exist, we will remove the join condition.

In the above process, we look into the database to fix the errors related to string format and minor syntax issues such as upper/lower cases but do not consider the semantics of contents. Therefore, we do not use database contents to facilitate the semantic understanding of the question or SQL generation. Some previous works [10, 13] also employ self-consistency techniques [45] for post-processing, which needs to execute the generated SQL in database before making the final output. Unlike such practices, we did not utilize the execution results of SQL in our approach.

| | |
|-------------|---|
| Instruction | <p>#### For the given question, use the provided tables, columns, foreign keys, and primary keys to fix the given SQLite SQL QUERY for any issues. If there are any problems, fix them. If there are no issues, return the SQLite SQL QUERY as is.</p> <p>#### Use the following instructions for fixing the SQL QUERY:</p> <ol style="list-style-type: none"> 1) Use the db_name values that are explicitly mentioned in the question. 2) Pay attention to the columns that are used for the JOIN by using the Foreign keys. 3) Pay attention to the columns that are used for the GROUP BY statement. 4) Pay attention to the columns that are used for the SELECT statement. |
| Schema | <p>#### Given the following database schema:</p> <pre>CREATE TABLE "continents" ("Contid" INTEGER PRIMARY KEY, "Continent" TEXT) CREATE TABLE "countries" ("Countryid" INTEGER PRIMARY KEY, "CountryName" TEXT, "Continent" INTEGER, FOREIGN KEY (Continent) REFERENCES continents(Contid)) CREATE TABLE "ca_makers" ("Id" INTEGER PRIMARY KEY, "Maker" TEXT, "FullName" TEXT, "Country" TEXT, FOREIGN KEY (Country) REFERENCES countries(Countryid)) CREATE TABLE "model_list" ("Modelid" INTEGER PRIMARY KEY, "Maker" INTEGER, "Model" TEXT UNIQUE, FOREIGN KEY (Maker) REFERENCES ca_makers(Id)) CREATE TABLE "car_names" ("Makeld" INTEGER PRIMARY KEY, "Model" TEXT, "Make" TEXT, FOREIGN KEY (Model) REFERENCES model_list (Modelid)) CREATE TABLE "cars_data" ("Id" INTEGER PRIMARY KEY, "MPG" TEXT, "Cylinders" INTEGER, "Edispl" REAL, "Horsepower" TEXT, "Weight" INTEGER, "Accelerate" REAL, "Year" INTEGER, FOREIGN KEY (Id) REFERENCES car_names (Makeld))</pre> |
| Question | <p>#### Question: Which model of the car has the minimum horsepower?</p> <p>#### SQLite SQL QUERY</p> <pre>SELECT model_list.Model FROM model_list JOIN car_names ON model_list.Modelid = car_names.Makeld JOIN cars_data ON car_names.Makeld = cars_data.Id ORDER BY cars_data.Horsepower ASC LIMIT 1;</pre> <p>#### SQLite FIXED SQL QUERY</p> <pre>SELECT</pre> |

Figure 7: The Template for Prompt-based Error Correction

We also develop a prompt-based method to correct errors in the generated SQL with one more iteration with LLMs in a zero-shot learning manner. An example is shown in Figure 7. The structure of this prompt is similar to that shown in Figure 2, i.e., it will include the instruction and schema information. In addition, the generated SQL is also included as part of the question, and the request is to ask LLM to correct the potential errors. In this process, we provide some guidelines in the format of explicit rules as hints for the LLM to make proper corrections, such as “Pay attention to whether every join condition is necessary” and “Use DESC and DISTINCT when needed”. The choice of guidelines could be realized by rule-based heuristics. For example, if the required contents could be selected from an original table without a join, we will apply the guideline related to join conditions. In the above example shown in Figure 7, we assume the second rule related to the join condition

could help the LLM to recognize the unnecessary join condition in the generated SQL and fix it via another prompt.

It is easy to see that the rule-based method is simple but has limited coverage. At the same time, prompt-based methods are expensive as they require another generation, but they could fix some complicated errors with well-designed instructions. To make good use of both, we develop a choice strategy to decide whether to use each of them in the following way: First, we apply all rule-based methods for correction. If they can find some errors and fix them, we will not continue applying prompt-based methods. In addition, since the prompt-based method aims to fix complicated errors, we only apply it when the case seems to be challenging. For example, in the Spider dataset [50], we apply the prompt-based method only for instances belonging to hard and extra categories. When such information is missing, we can look into the demonstration examples in the original prompt, e.g., if there are examples with join conditions between more than 2 tables or conjunction operations, we will apply the prompt-based method.

4 EVALUATION

4.1 Experiment Setup

Table 1: The statistics of datasets

| Dataset | # Queries | # Databases | # Tables |
|----------------|-----------|-------------|----------|
| Spider (train) | 8,659 | 146 | 795 |
| Spider (dev) | 1,034 | 20 | 81 |
| Spider (test) | 2,147 | 40 | 180 |
| BIRD (train) | 9,428 | 69 | 524 |
| BIRD (dev) | 1,533 | 11 | 81 |

4.1.1 Datasets. We mainly conducted experiments on the benchmarking dataset Spider [50], which is widely used in previous studies about Text-to-SQL. We reported results on dev and test sets, which are released on the official website ². The instances in the training set of Spider are used as the candidate of demonstration examples for few-shot learning and report the results on both the dev and test sets. In addition, we also evaluated on the BIRD [24] dataset, which is recently proposed to evaluate the efficiency of the generated SQLs. Since our work focused on improving the effectiveness rather than efficiency of Text-to-SQL tasks, we only report results regarding accuracy but not the Valid Efficiency Score, which evaluates whether the generated SQL queries are optimized. The detailed statistics of these datasets are shown in Table 1.

4.1.2 Evaluation Metrics. Following the practice of previous studies [53], we use two metrics to evaluate the accuracy of the proposed solutions: Exact Set Match (EM) and Execution Match (EX) accuracy. Exact Set Match accuracy, which is also known as logical form accuracy, measures the matched SQL keywords between the predicted SQL query and the corresponding ground truth. Execution Match accuracy requires executing the generated SQL in a real database system and comparing the execution result with that of the gold standard SQL. It provides a more precise estimate of the model’s

performance since multiple valid SQL statements may exist for a single question. Furthermore, we also report the cost on Spider datasets by examining the total number of tokens in the prompts.

4.1.3 Baseline Methods. We primarily choose the following existing solutions as baseline methods to compare with:

DAIL-SQL [13] is the latest prompt-based method that explores a wise combination of prompt template and demonstration selection methods to improve the overall performance of LLMs.

DIN-SQL [31] utilizes a chain-of-thought strategy to divide the text-to-SQL problem into 3 stages and conduct prompts for each of them, respectively.

Augment [29] proposes a schema-related knowledge augmentation method to improve the prompt construction process to obtain high-quality SQL based on LLM.

CatSQL [11] is a template-filling based method that achieves the best performance in that category of works.

Graphix-T5 [22] constructs a graph to model the interaction between the question and database schema and then incorporates such information in the fine-tuning process of the decoder. It is the up-to-date one in the category of machine-translation-based methods.

Besides, we also include the results of earlier representative works, such as PICARD [38], RASAT [32], RYANSQL [7], LGESQL [5], SmBoP [35] and RESDSQL [22] in the comparison. We directly cite the numbers from the original papers and the leader board for all methods.

4.1.4 Environment. We implemented all proposed methods in Python. All experiments are run on a server with configurations similar to those of a g5.12xlarge AWS EC2 machine, which has one AMD EPYC 7R32 48-core processor and 192GB RAM. We reported the results of prompt over OpenAI APIs for both GPT-4 (gpt-4-0613) and GPT-3.5 (gpt-3.5-turbo-0125) for the SQL generation. Due to the budget constraint, we only use GPT-3.5 for the experiments with large-scale prompts, e.g., the study about different numbers of demonstrations in Figure 9. Otherwise, the results for all LLM-based solutions will be based on GPT-4 if there is no additional explanation.

4.2 Comparative Performance Measurement

The main results on the Spider dataset are shown in Table 2. We have the following observations: First of all, LLM-based solutions achieved better performance in EX. This is due to the power of LLM in understanding the input question and generating SQL accordingly. At the same time, the results of EM are not as good as PLM-based methods. The reason is that LLM-based solutions generate the SQL according to the semantics of the question based on the inherited knowledge gained in the pre-training process, while PLM-based methods learn the syntax of SQL queries from the training set, which has a syntax structure more similar to those in the dev and test sets. Nevertheless, as shown in recent studies [11, 13, 20, 22, 29, 31], EX is a more critical metric in evaluating the main results as it is more closely related to the performance in real scenarios. Therefore, it is safe to claim the superiority of LLM-based methods only based on the EX results.

²<https://yale-lily.github.io/spider>

Table 2: Main Results on the Spider Dataset. “-” means the corresponding result is not available in the original paper or any public leader board.

| Method | Dev | | Test | |
|-------------------|-------------|-------------|-------------|-------------|
| | EM (%) | EX (%) | EM (%) | EX (%) |
| RYANSQL | 66.4 | 58.2 | - | - |
| LGESQL | 75.1 | 34.8 | 72.0 | - |
| SmBoP | 74.7 | 77.9 | 71.1 | 69.5 |
| PICARD | 75.5 | 79.3 | - | 75.1 |
| RASAT | 74.7 | 80.5 | 70.6 | 75.5 |
| Graphix-T5 | 77.1 | 81.0 | 74.0 | 77.6 |
| RESDSQL | 80.5 | 84.1 | 72.0 | 79.9 |
| CatSQL | 80.6 | 83.7 | 73.9 | 78.0 |
| Augment (GPT-4) | - | 84.1 | - | - |
| DIN-SQL (GPT-4) | 60.1 | 74.2 | 60.0 | 85.3 |
| DAIL-SQL (GPT-4) | 71.9 | 82.4 | - | 86.2 |
| MageSQL (GPT-3.5) | 58.5 | 81.6 | 55.7 | 80.5 |
| MageSQL (GPT-4) | 69.7 | 87.4 | 67.2 | 86.8 |

In addition, MageSQL performs better than other LLM-based solutions³. The reason is that MageSQL proposed effective demonstration selection techniques that could customize the demonstration examples for each question. In this way, it would provide useful signals for different questions to the LLM. At the same time, our error correction techniques could help fix various errors in the LLM output. In this way, errors due to lack of sufficient context information could be avoided.

Table 3: Performance Breakdown based on Difficulty on Spider (EX %)

| Split | Method | Easy | Normal | Hard | Extra | Overall |
|-------|-------------------|-------------|-------------|-------------|-------------|-------------|
| Dev | MageSQL (GPT-3.5) | 93.1 | 86.8 | 70.7 | 62.0 | 81.6 |
| | MageSQL (GPT-4) | 96.4 | 90.8 | 82.2 | 70.5 | 87.4 |
| | CatSQL (GPT-4) | 95.6 | 88.3 | 74.7 | 62.7 | 83.7 |
| | DIN-SQL (GPT-4) | 91.1 | 79.8 | 64.9 | 43.4 | 74.2 |
| Test | MageSQL (GPT-3.5) | 91.9 | 83.5 | 71.7 | 69.5 | 80.5 |
| | MageSQL (GPT-4) | 92.3 | 89.6 | 82.1 | 78.7 | 86.8 |

We show the results of performance breakdown based on the query difficulty in Table 3. Since very few previous studies reported such results, we only include the comparison with CatSQL [11] and DIN-SQL [31] on the dev set. We can see that compared with previous studies, MageSQL has more improvement in the harder cases. Compared with the template filling-based method CatSQL, MageSQL could take advantage of the power of LLM in understanding the question and code generation to improve the overall performance. The performance of MageSQL is much better than

³DAIL-SQL with self-consistency could reach the results of 86.6. However, it requires running the prompt multiple times and executing the generated SQL in the database before the final output. We report the result without self-consistency for a fair comparison.

another LLM-based method DIN-SQL in hard and extra-hard categories. The reason might be the useful insights provided by properly selected demonstrations.

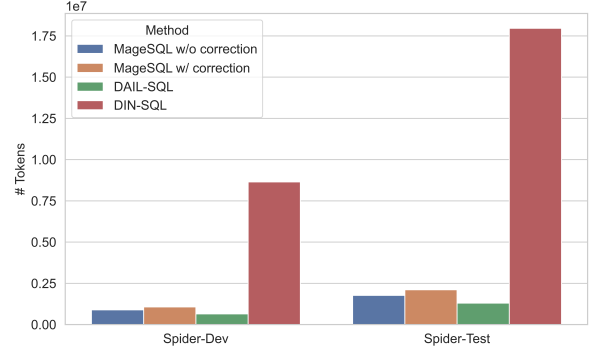


Figure 8: Cost of Different LLM-based Methods with 5-shot learning on the Spider Dataset.

Finally, we report the cost of MageSQL and other recent LLM-based methods, DIN-SQL and DAIL-SQL, in Figure 8. Based on the official OpenAI pricing mechanism⁴, we use the total number of tokens in all prompts as the evaluation metric. We can see that DIN-SQL involves the most overhead in cost since it adopts the chain-of-thought method and requires 3 prompts for each instance. Our method requires an extra prompt to generate the SQL to find examples with similar graph embeddings, though such prompts are relatively short since there is no demonstration. Thus, the cost (even without error correction) is slightly higher than that of DAIL-SQL. Meanwhile, error correction did not introduce much additional cost to our method. The reason could be due to the strategy scheduling efforts shown in Section 3.4 that only send the challenging cases to the prompt-based error correction.

4.3 Impact of MageSQL Design Choices

Table 4: Results on Different Demonstration Selection Strategies (EX %).

| Method | Spider-Dev | Spider-Test |
|---------------------|-------------|-------------|
| Zero-shot Learning | 77.6 | 77.9 |
| Random | 81.8 | 81.9 |
| Hardness | 83.5 | 83.1 |
| Question Similarity | 84.0 | 84.4 |
| Struct-Tree | 84.9 | 86.6 |
| Struct-Graph | 87.4 | 86.8 |

Next, we conducted more experiments to analyze the effects of our proposed techniques. We will use Execution Accuracy (EX) as the evaluation metric because it is more appropriate for LLM-based solutions.

⁴<https://openai.com/api/pricing/>

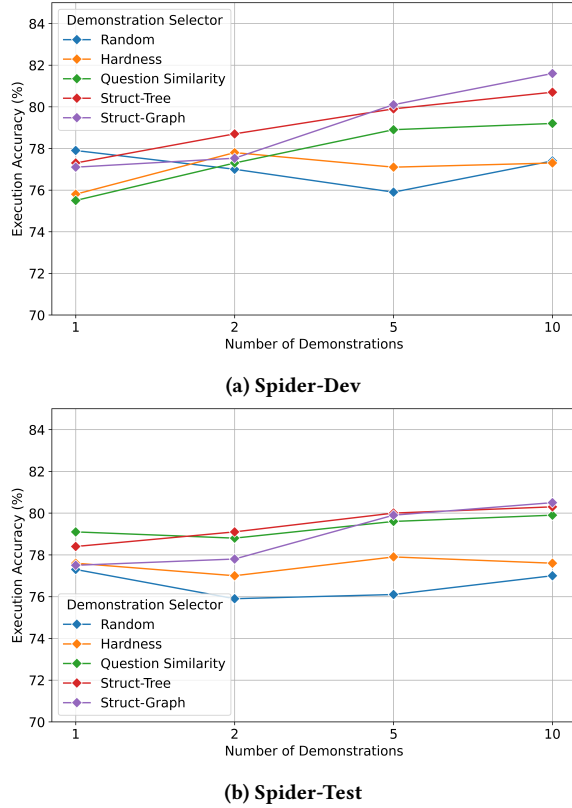


Figure 9: Effect of numbers of demonstrations for Spider datasets based on GPT-3.5. The result of zero-shot learning on the dev and test set is 75.4 and 76.0, respectively.

We first look at the effect of different demonstration selection strategies in Table 4. The methods Random, Hardness, Question Similarity and Struct-Tree were previously introduced in Section 3.2; while Struct-Graph is the graph embedding-based method in Section 3.3. Since different strategies might need different numbers of demonstration examples to achieve the best performance in few-shot learning, and such numbers don’t differ much (no more than 10-shot), we report the best performance of all methods that might not have the same number of demonstrations. Generally speaking, we observe that Random performs worst, and in fact, it is close to a zero-shot learning setting. This clearly illustrates that bad demonstrations might harm the results in some cases. Among all the methods, Struct-Graph achieves the best result since it could provide useful examples for some difficult instances to help LLM generate the corresponding SQL. Although Hardness can reach similar objectives, its selection criteria are too heuristic and might not be able to find proper examples.

We then investigate the effect of a number of demonstrations. As shown in Figure 9, the results of most methods tend to be better with more examples. The exceptions are in the Random and Harness cases. The reason could be that they both include the process of randomly selecting examples from a set of candidates and thus might not always select high-quality ones. We also tried to include

more than 10 examples as demonstrations. However, the results do not improve. Therefore, we stop with the maximum number of examples as 10.

We also show the effect of error correction methods with execution accuracy (EX) as the metric. The results are as follows: on Spider-dev, the result of execution accuracy without and with error correction is 84.5 and 87.4, respectively. On the Spider-test, execution accuracy without and with error correction is 84.7 and 86.8, respectively. With the help of error correction, we achieve up to 2.9% performance gain on all datasets. It illustrates that the error correction mechanism could help address some errors from the SQL generated from the initial prompt. The reason could be that the pre-defined instructions could provide more useful insights for the second prompt to generate the correct query. Besides, the rule-based approach could also help fix some instances where the semantics are correct but fail in execution just because of minor issues, e.g., different letter cases in predicate values, extra symbols like quota, etc.

4.4 Results on the BIRD Dataset

| | |
|----------------|--|
| Instruction | ### Complete sqlite SQL query only and with no explanation. |
| Demonstrations | ### Some example pairs of question and corresponding SQL query are provided based on similar problems: ### Answer the following question: List down the game platform IDs of games with a region ID of 1. SELECT T.game_platform_id FROM region_sales AS T WHERE T.region_id = 1 ### Answer the following question: List down all of the film IDs with highest rental duration. SELECT film_id FROM film WHERE rental_duration = (SELECT MAX(rental_duration) FROM film) |
| Context | ### Given the following database schema: CREATE TABLE account (account_id INTEGER default 0 not null primary key, district_id INTEGER default 0 not null, frequency TEXT not null, date DATE not null, foreign key (district_id) references district (district_id)) CREATE TABLE card (card_id INTEGER default 0 not null primary key, disp_id INTEGER not null, type TEXT not null, issued DATE not null, foreign key (disp_id) references disp (disp_id)) |
| Evidence | ### Given the following evidence: 'POPLATEK MESICNE' stands for monthly issuance |
| Question | ### Answer the following question: List the account IDs with monthly issuance of statements. SELECT |

Figure 10: The Prompt Template for BIRD Dataset.

Next we conduct some additional experiments on the BIRD [23] datasets to provide more insights about our proposed work. Compared with Spider, there are more tables in each database of the BIRD dataset and the model needs to accurately identify the relevant tables to answer a question. Meanwhile, the BIRD dataset provided additional “evidence” information, which are paragraphs of descriptions about databases and tables to assist disambiguation in the questions with such external knowledge. Therefore, it is essential to include them in the prompt template to help with question understanding, as previous studies on the BIRD dataset have done. To this end, we slightly modify the previous prompt template to satisfy the need of BIRD as shown in Figure 10. Specifically, we add a section of “Evidence” (yellow box) at the end of Context in the prompt to accommodate such information.

Here we use the representative previous studies compared in the recent work SuperSQL [19] as baseline methods. The results of baseline methods are copied from [19]. For a method with multiple

Table 5: Results on the BIRD-dev dataset with performance breakdown based on difficulty levels (EX %). All methods are based on GPT-4

| Method | Simple | Moderate | Challenging | Overall |
|----------|--------------|-------------|--------------|--------------|
| RESDSL | 53.5 | 33.3 | 16.7 | 43.9 |
| C3 | 58.9 | 38.5 | 31.9 | 50.2 |
| DAIL-SQL | 63.0 | 45.6 | 43.1 | 55.9 |
| CodeS | 65.8 | 48.8 | 42.4 | 58.5 |
| SuperSQL | 66.9 | 46.5 | 43.8 | 58.5 |
| MageSQL | 68.54 | 48.06 | 51.03 | 60.69 |

variants, we reported the one with the best results. For example, RESDSL [20] is corresponding to RESDSL-3B; DAIL-SQL [13] is the version with Self-Consistency (SC); and CodeS [21] is corresponding to SFT CodeS-15B. The results shown in Table 5 illustrated that MageSQL achieved the best overall performance among all methods. Specifically, it outperformed the state-of-the-art method SuperSQL [19] by 2.19% in execution accuracy. Compared with the Spider dataset, BIRD is more challenging due to its huge database volumes and much larger number of numbers in a database. MageSQL could alleviate such issues with the help of high-quality demonstration examples and the ability to fix minor errors in the model output. We observe that the advantage of MageSQL over other baseline methods is more obvious in the difficulty level of “Challenging” which is consistent with that on the Spider dataset.

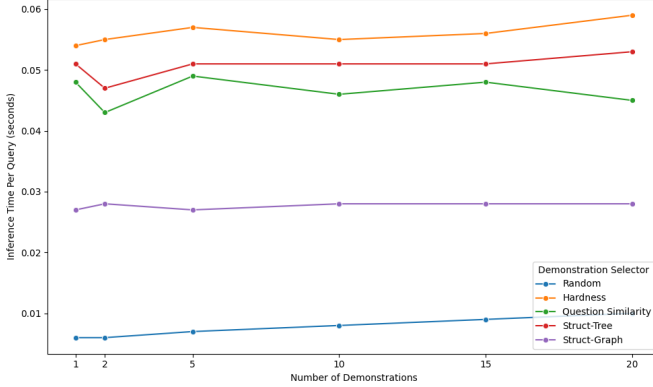


Figure 11: Average Time of Demo Selection on BIRD-dev.

We also report the average execution time per instance of different demonstration selection strategies in Figure 11. We can see that our proposed selection strategies are light weighted, where the worse method only requires less than 0.06s for one instance on average. This overhead is trivial in the overall pipeline. Besides, the Struct-Graph method also requires a training process with around 1 hour. However, it is an offline process and only needs to be executed one time before the selection process.

4.5 Error Analysis

We further conducted an in-depth error analysis to provide useful insights about our techniques. To begin with, we compiled statistics

on the instances in which our proposed solution failed in the Spider dataset even after error correction. Based on the practice of previous studies [10, 11, 31], we made the category of errors as following:

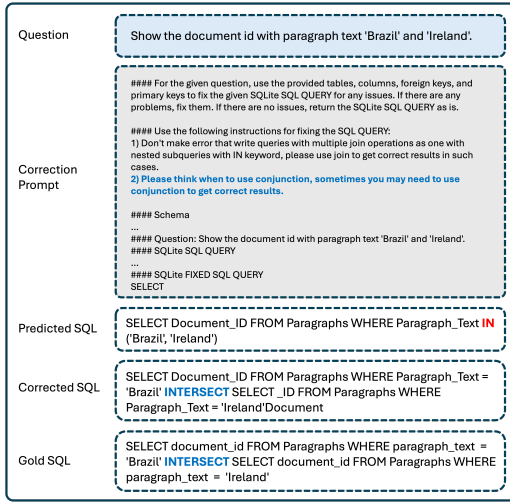
- Syntax: There are syntax errors, and the generated SQL cannot be executed.
- Structure: The generated SQL failed to identify or make obvious errors in the structure of a query, such as those with multi-way join and conjunction.
- Schema: The error are related to the schema information of database.
- Name and Semantics: The error is related to the semantics of table/attribute names or values in predicates.
- Aggregation: The error is related to aggregations.

Table 6: Error Analysis on Spider Dataset (%). The number is the percentage in all incorrect instances but not all instances in the dataset.

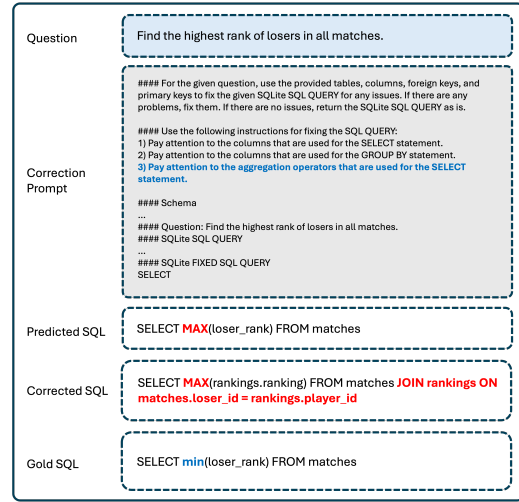
| Error Category | Dev | Test |
|--------------------|------|------|
| Syntax | 9.6 | 10.1 |
| Structure | 58.9 | 20.8 |
| Schema | 45.5 | 25.3 |
| Name and Semantics | 28.2 | 43.8 |
| Aggregation | 39.7 | 18.1 |

The results of statistics are shown in Table 6. We recognize that one incorrect instance could involve multiple types of errors. Therefore, the overall number could exceed 100% in each dataset. We can see that most errors in the Dev set come from the Structure and Schema categories, which correspond to the instances in the hard and extra categories. At the same time, the challenges in the Test set mainly come from Name and Semantics, where many cases require the LLM to understand not only the question but also the semantics of table and attribute names. In such cases, errors in the “Name and Semantics” category always happened together with those in the “Schema” one.

Next, we conduct a case study about our prompt-based error correction method in Figure 12. The detailed schema and query in correction prompts are omitted due to space limitations. Figure 12(a) illustrates a scenario where the prompt-based method can successfully identify and fix the error. We can see that the initial output of LLM has errors in identifying the need to use conjunctions. We apply a prompt with the template shown in Figure 7 and utilize the guidelines “Please think when to use a conjunction; sometimes you may need to use a conjunction to get correct results”. Then, such errors could be fixed with another round of prompts. At the same time, the prompt-based strategy may also fail in some instances, as illustrated in Figure 12(b). In this example, the generated SQL incorrectly interprets the semantics of “highest rank”. Despite adhering to the guidelines that emphasize appropriate aggregation operators in the SELECT statement, the final output is still an erroneous SQL statement. This error might stem from the LLM’s limitations in handling the semantic nuances of the query. We are thrilled to continue exploring how to fix such issues via prompt or advanced rules in future work.



(a) Successful example of the prompt-based correction method



(b) Failure example of the prompt-based correction

Figure 12: Examples of Error Correction with Prompt.

5 RELATED WORK

The Text-to-SQL problem has been well-explored for many years. Earlier studies [16, 36] first parsed the natural language question into intermediate results and then developed different kinds of rules to map it into the abstract syntax tree of SQL so as to generate the final query. The limitation of such methods is that they always perform poorly when adapted to new domains. To address such issues, another category of studies employed deep learning techniques, converting the Text-to-SQL problem into a machine translation task and train deep learning models to generate SQL statements. Based on the different ways of decoding, the solutions can be categorized into sequence [15, 38], tree [32, 35] and graph [3, 5, 22] based ones. With the advances of PLM, recent studies developed the solutions by pre-training a language model for structured data to support various tasks including text-to-SQL, such as TAPEX [27] and GraPPa [51]. Meanwhile, another category of studies first developed a sketch template of SQL and then use decoder based models to fill the empty slots in the template [7, 11, 12]. This approach could avoid generating invalid SQL queries while fully utilizing the powerful decoders.

Recent efforts mainly aimed at leveraging the power of LLMs to understand the natural language question and generate the SQL accordingly. Due to their profound capabilities, LLM-based solutions have significantly outperformed previous methods and achieved state-of-the-art performance. C3 [10] proposed a precise prompt instruction for the zero-shot setting. Rajkumar et al. [33] made an investigation of prompt strategies over different kinds of LLMs. Nan et al. [29] explored different prompt templates for both zero-shot and few-shot settings. DIN-SQL [31] employed the chain-of-thought reasoning strategy and divided the text-to-SQL problem into 3 stages to solve issues from different aspects in each stage respectively. DAIL-SQL [13] conducted an empirical study on the different combinations of prompt instructions and demonstration selection strategies, which overlaps with our work but focuses on different

aspects of the problem. SuperSQL [19] explored the combination of different components in prompt templates and summarized optimal solutions for different tasks. CodeS [21] aims at fine tuning smaller language models instead of constructing prompts, which has a different objective.

Another line of studies lie in the aspect of benchmarking the text-to-SQL tasks. There are two categories of datasets: general purposed and domain specific ones. The general purposed works aimed at building cross-domain datasets that have a broad coverage. Examples include WikiSQL [54], Spider [50], KaggleDBQA [18] and BIRD [24]; While the domain specific works focused on developing datasets for a specific application domain, such as Yelp and IMDB [48], FINSQL [52] and BookSQL [17]. In this work, we focused on developing general solution for the text-to-SQL problem and thus conducted evaluation over the general purposed benchmarking datasets.

6 CONCLUSION

In this paper, we conducted a systematic study of the Text-to-SQL problem. We proposed MageSQL, a novel framework that aimed at improving the prompt engineering process over LLM so as to help generate high-quality SQL statements as the solution. To this end, we proposed technical contributions from two aspects: (1) develop two effective demonstration selection strategies based on the structure and semantics of SQL queries to improve the in-context learning process; (2) propose an error correction module that could find and correct the potential errors in the output of LLMs. Experimental results on public benchmarking datasets showed that our proposed methods could obviously improve the overall performance compared with previous solutions.

REFERENCES

- [1] Nikolaus Augsten, Michael H. Böhlen, and Johann Gamper. 2010. The *pg*-gram distance between ordered labeled trees. *ACM Trans. Database Syst.* 35, 1 (2010), 4:1–4:36.

- [2] Philip Bille. 2005. A survey on tree edit distance and related problems. *Theor. Comput. Sci.* 337, 1-3 (2005), 217–239.
- [3] Ben Bogin, Jonathan Berant, and Matt Gardner. 2019. Representing Schema Structure with Graph Neural Networks for Text-to-SQL Parsing. In *ACL*. 4560–4565.
- [4] Tom B. Brown and et al. 2020. Language Models are Few-Shot Learners. In *NeurIPS*.
- [5] Ruisheng Cao, Lu Chen, Zhi Chen, Yanbin Zhao, Su Zhu, and Kai Yu. 2021. LGE SQL: Line Graph Enhanced Text-to-SQL Model with Mixed Local and Non-Local Relations. In *ACL/IJCNLP*. 2541–2555.
- [6] Mark Chen and et al. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021).
- [7] DongHyun Choi, Myeongcheol Shin, EungGyun Kim, and Dong Ryeol Shin. 2021. RYANS SQL: Recursively Applying Sketch-based Slot Fillings for Complex Text-to-SQL in Cross-Domain Databases. *Comput. Linguistics* 47, 2 (2021), 309–332.
- [8] Aakanksha Chowdhery and et al. 2023. PaLM: Scaling Language Modeling with Pathways. *J. Mach. Learn. Res.* 24 (2023), 240:1–240:113.
- [9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*. 4171–4186.
- [10] Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Lu Chen, Jinshu Lin, and Dongfang Lou. 2023. C3: Zero-shot Text-to-SQL with ChatGPT. *CoRR* abs/2307.07306 (2023).
- [11] Han Fu, Chang Liu, Bin Wu, Feifei Li, Jian Tan, and Jianling Sun. 2023. CatSQL: Towards Real World Natural Language to SQL Applications. *Proc. VLDB Endow.* 16, 6 (2023), 1534–1547.
- [12] Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R. Woodward, John H. Drake, and Qiaofu Zhang. 2021. Natural SQL: Making SQL Easier to Infer from Natural Language Specifications. In *Findings of EMNLP*. 2030–2042.
- [13] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. *Proc. VLDB Endow.* 17, 5 (2024), 1132–1145.
- [14] Zhenyu Hou, Xiao Liu, Yukuo Cen, Yuxiao Dong, Hongxia Yang, Chunjie Wang, and Jie Tang. 2022. GraphMAE: Self-Supervised Masked Graph Autoencoders. In *ACM SIGKDD*. 594–604.
- [15] Yiqun Hu and et al. 2023. Importance of Synthesizing High-quality Data for Text-to-SQL Parsing. In *ACL*. 1327–1343.
- [16] Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. 2020. Natural language to SQL: Where are we today? *Proc. VLDB Endow.* 13, 10 (2020), 1737–1750.
- [17] Rahul Kumar, Amar Raja Dibbu, Shrutendra Harsola, Vignesh Subrahmaniam, and Ashutosh Modi. 2024. BookSQL: A Large Scale Text-to-SQL Dataset for Accounting Domain. In *NAACL*. 497–516.
- [18] Chia-Hsuan Lee, Oleksandr Polozov, and Matthew Richardson. 2021. KaggleD-BQA: Realistic Evaluation of Text-to-SQL Parsers. In *ACL/IJCNLP*. 2261–2273.
- [19] Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024. The Dawn of Natural Language to SQL: Are We Fully Ready? *Proc. VLDB Endow.* 17, 11 (2024), 3318–3331.
- [20] Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023. RESDSQL: Decoupling Schema Linking and Skeleton Parsing for Text-to-SQL. In *AAAI*. 13067–13075.
- [21] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024. CodeS: Towards Building Open-source Language Models for Text-to-SQL. *Proc. ACM Manag. Data* 2, 3 (2024), 127.
- [22] Jinyang Li, Binyuan Hui, Reynold Cheng, Bowen Qin, Chenhao Ma, Nan Huo, Fei Huang, Wenyu Du, Luo Si, and Yongbin Li. 2023. Graphix-T5: Mixing Pre-trained Transformers with Graph-Aware Layers for Text-to-SQL Parsing. In *AAAI*. 13076–13084.
- [23] Jinyang Li, Binyuan Hui, Ge Qu, Jiayi 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).
- [24] Jinyang Li, Binyuan Hui, Ge Qu, Jiayi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chen-Chuan Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM Already Serve as A Database Interface? A Big Bench for Large-Scale Database Grounded Text-to-SQLs. In *NeurIPS*.
- [25] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2022. What Makes Good In-Context Examples for GPT-3? In *The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures, DeLio@ACL*. 100–114.
- [26] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv.* 55, 9 (2023), 195:1–195:35.
- [27] Qian Liu, Bei Chen, Jiaqi Guo, Morteza Ziyadi, Zeqi Lin, Weizhu Chen, and Jian-Guang Lou. 2022. TAPEX: Table Pre-training via Learning a Neural SQL Executor. In *ICLR*.
- [28] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. 2022. Fantastically Ordered Prompts and Where to Find Them: Overcoming Few-Shot Prompt Order Sensitivity. In *ACL*. 8086–8098.
- [29] Linyong Nan, Yilun Zhao, Weijin Zou, Narutatsu Ri, Jaesung Tae, Ellen Zhang, Arman Cohan, and Dragomir Radev. 2023. Enhancing Text-to-SQL Capabilities of Large Language Models: A Study on Prompt Design Strategies. In *Findings of EMNLP*. 14935–14956.
- [30] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023).
- [31] Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. In *NeurIPS*.
- [32] Jiexing Qi, Jingyao Tang, Ziwei He, Xiangpeng Wan, Yu Cheng, Chenghu Zhou, Xinbing Wang, Quanshi Zhang, and Zhouhan Lin. 2022. RASAT: Integrating Relational Structures into Pretrained Seq2Seq Model for Text-to-SQL. In *EMNLP*. 3215–3229.
- [33] Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. 2022. Evaluating the Text-to-SQL Capabilities of Large Language Models. *CoRR* abs/2204.00498 (2022).
- [34] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *EMNLP-IJCNLP*. 3980–3990.
- [35] Ohad Rubin and Jonathan Berant. 2021. SmBoP: Semi-autoregressive Bottom-up Semantic Parsing. In *NAACL-HLT*. 311–324.
- [36] Diptikalyan Saha, Avriella Floratos, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R. Mittal, and Fatma Özcan. 2016. ATHENA: An Ontology-Driven System for Natural Language Querying over Relational Data Stores. *Proc. VLDB Endow.* 9, 12 (2016), 1209–1220.
- [37] Timo Schick and et al. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *NeurIPS* 2023.
- [38] Torsten Scholach, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *EMNLP*. 9895–9901.
- [39] Chen Shen, Jin Wang, Sajjadur Rahman, and Eser Kandogan. 2024. Demonstration of a Multi-agent Framework for Text to SQL Applications with Large Language Models. In *CIKM*. ACM, 5280–5283.
- [40] Kihyuk Sohn. 2016. Improved Deep Metric Learning with Multi-class N-pair Loss Objective. In *Advances in Neural Information Processing Systems*. 1849–1857.
- [41] Fan-Yun Sun, Jordan Hoffmann, Vikas Verma, and Jian Tang. 2020. InfoGraph: Unsupervised and Semi-supervised Graph-Level Representation Learning via Mutual Information Maximization. In *ICLR*.
- [42] Hugo Touvron and et al. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR* abs/2302.13971 (2023).
- [43] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation Learning with Contrastive Predictive Coding. *CoRR* abs/1807.03748 (2018).
- [44] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *ICLR*.
- [45] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *ICLR*.
- [46] Jason Wei and et al. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *NeurIPS*.
- [47] Qingyun Wu and et al. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation Framework. *CoRR* abs/2308.08155 (2023).
- [48] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 63:1–63:26.
- [49] Yuning You, Tianlong Chen, Yongduo Sui, Ting Chen, Zhangyang Wang, and Yang Shen. 2020. Graph Contrastive Learning with Augmentations. In *NeurIPS*.
- [50] Tao Yu and et al. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *EMNLP*. 3911–3921.
- [51] Tao Yu, Chien-Sheng Wu, Xi Victoria Lin, Bailin Wang, Yi Chern Tan, Xinyi Yang, Dragomir R. Radev, Richard Socher, and Caiming Xiong. 2021. GraPPa: Grammar-Augmented Pre-Training for Table Semantic Parsing. In *ICLR*.
- [52] Chao Zhang, Yuren Mao, Yijiang Fan, Yu Mi, Yunjun Gao, Lu Chen, Dongfang Lou, and Jinshu Lin. 2024. FinSQL: Model-Agnostic LLMs-based Text-to-SQL Framework for Financial Analysis. In *Companion of SIGMOD/PODS*. ACM, 93–105.
- [53] Ruiqi Zhong, Tao Yu, and Dan Klein. 2020. Semantic Evaluation for Text-to-SQL with Distilled Test Suites. In *EMNLP*. 396–411.
- [54] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *CoRR* abs/1709.00103 (2017).
- [55] Yanqiao Zhu, Yichen Xu, Qiang Liu, and Shu Wu. 2021. An Empirical Study of Graph Contrastive Learning. In *NeurIPS Datasets and Benchmarks*.