SA-SQL: A Schema-Aligned Collaborative Framework for Text-to-SQL

Anonymous ACL submission

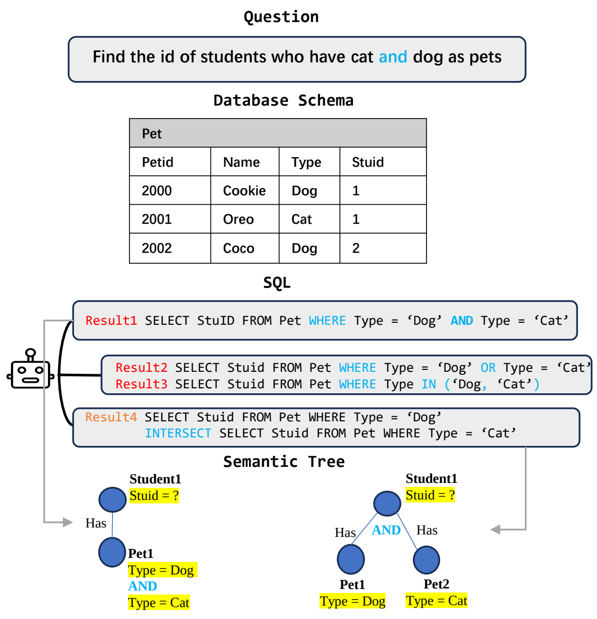
Abstract

Text-to-SQL is a important task for database-related applications, which automatically translates users' questions into SQL queries for database. Recently, the large language models (LLMs) has demonstrated promising performances in the task. However, most existing methods do well in translating the question semantics into the SQL, but cannot correctly align the SQL with the database schema (e.g., the primary keys), which leads to several errors in the generated SQL. To tackle the problem, we design SA-SQL, a novel schema-aligned framework for Text-to-SQL task following the draft-and-correction manner. First, we select the question-relevant tables and columns from the whole schema with a selector to exclude the noise for generating a concise SQL and reduce the input length. Next, based on the selected tables and columns, we generate a draft of the SQL with a generator to capture the question semantics. Last, considering the alignment between the SQL and the database schema, we design a corrector to detect and correct the possible errors in the draft. We conduct extensive experiments on two widely-used dataset for Text-to-SQL, and our framework has achieved a state-of-the-art execution accuracy of 89.8% on the Spider dataset, and 59.8% on the BIRD dataset, which proves the effectiveness of our framework.

1 Introduction

Text-to-SQL is a classic task in natural language processing, which aims to automatically translate users' questions in natural language into SQL queries for databases. It provides a natural language interface for databases, which benefits various applications related to database and tabular data(Deng et al., 2022).

Recently, the large language models (LLMs) has become the latest developments in Text-to-SQL tasks(Liu et al., 2023; Gao et al., 2023; Nan et al., 2023), which has shown promising performances due to their strong reasoning and in-context learning ability(Brown et al., 2020). Following the in-context learning paradigm, recent researches provide the LLMs with chains of reasoning(Sun et al., 2023), i.e., examples that contain more sub-steps relevant to the questions, making LLMs easily parse key information from the question and translate into SQL queries.



Find the id of students who have cat and dog as pets

Pet			
Petid	Name	Type	Stuid
2000	Cookie	Dog	1
2001	Oreo	Cat	1
2002	Coco	Dog	2

```
Result1 SELECT StuID FROM Pet WHERE Type = ‘Dog’ AND Type = ‘Cat’
Result2 SELECT Stuid FROM Pet WHERE Type = ‘Dog’ OR Type = ‘Cat’
Result3 SELECT Stuid FROM Pet WHERE Type IN (‘Dog, ‘Cat’)
Result4 SELECT Stuid FROM Pet WHERE Type = ‘Dog’
        INTERSECT SELECT Stuid FROM Pet WHERE Type = ‘Cat’
```

Figure 1: Four possible results for the same question on GPT3.5-turbo with three different semantics. The bottom is the semantic tree corresponding to result1 and result4.

However, although existing LLM-based methods has performed well in translating the question semantics into SQL queries(Gao et al., 2023; Nan et al., 2023; Dong et al., 2023), they seldom take full advantages of the database schema (e.g., the primary key and foreign keys of each table in the database), and thus easily make errors in aligning with the database schema in complex SQL generation. For example, in Figure 1, all generated SQLs capture the semantics of "and" in the question by “AND”, “OR”, “IN” and “INTERSECT”, but only the last one correctly matches the schema that one Stuid only corresponds to one Type in one record, but several Type in different records.

Moreover, we conduct a comprehensive analysis on the wrong SQLs from the LLM (see Section 3.2), and summarize major alignment errors as follows. (1) **Group Error**: the LLM may use incorrect column in the GROUP clause to identify each entity. For example, given the question “Show the names of students who have likes, and numbers of pets for each”, the LLM mistakenly use “student.name” in the GROUP clause instead of the primary key “student.stuid” to represent each student for counting pet number. (2) **Connect Error**: for questions with multiple conditions, the LLM may use the wrong logical connector for the conditions, i.e., AND or INTERSEC, OR or UNION, and NOT or EXCEPT as shown in Figure 1. (3) **Value Error**: without consideration of the database content, the LLM may represent the entity with terms that do not appear in the database (e.g., ‘usa’ or ‘the united states’).

To tackle these problems, we design a novel **S**chema-**A**ligned framework (SA-SQL) based on LLMs for Text-to-SQL tasks following the draft-and-correction manner. The framework is composed of three key components, the *Selector*, *Generator* and *Corrector*. Specifically, given the question and database schema, the Selector first filters the irrelevant tables and columns from the schema based on the question semantics to exclude the noise for generating a concise SQL and reduce input length. Next, based on the selected tables and columns, the Generator generates a SQL as the draft, which captures the question semantics and follows the SQL syntax. Last, considering the alignment errors mentioned above, the Corrector detects the possible errors in the draft, and corrects them to generate the final SQL as the output. We implement the Corrector based on the three major errors we found, and it can be easily extended for other errors. To verify the effectiveness of the proposed framework, we conduct extensive experiments on two widely-used datasets for Text-to-SQL task, i.e., the Spider and the BIRD, and our framework has achieved a state-of-the-art execution accuracy on the two datasets.

2 Related Works

2.1 Large Language Models on Text-to-SQL

Recently, significant progress has been made in using LLMs for the text-to-SQL task, with several methodologies proposed to enhance their capabilities. In the early stages of the emergence of large language models, research efforts were primarily focused on designing high-quality prompts to better exploit the potential of LLMs for SQL generation. For example, SQLPrompt(Tai et al., 2023) systematically studied how to enhance LLMs’ reasoning ability through chain-of-thought style prompting, including the original chain-of-thought prompting and least-to-most prompting. Additionally, DAIL-SQL (Gao et al., 2023) systematically examined prompt engineering for LLM-based Text-to-SQL methods, including question representations, prompt components, example selections, and example organizations.

Subsequent research has increasingly emphasized the development of multi-stage refined frameworks that revolve around large language models. These frameworks aim to simplify databases, generate SQL, verify queries, and integrate answers effectively. For instance, C3-SQL (Dong et al., 2023) proposed a zero-shot Text-to-SQL approach based on ChatGPT, which enhances performance through the use of appropriate input and calibration hints, as well as a self-consistency strategy. Additionally, DIN-SQL(Pourreza and Rafiei, 2023) introduced a query classification and decomposition module that breaks down the text-to-SQL task into smaller subproblems and solves them using prompting techniques. Moreover, StructGPT(Jiang et al., 2023) aims to enable LLMs to reason over structured data, such as databases, with a framework consisting of a set of specialized interfaces that allow LLMs to access and filter structured data, and an iterative reading-then-reasoning procedure.

3 Preliminaries

3.1 Problem Definition

3.1.1 Database Schema

A relational database is denoted as $\mathcal{D}$. The database schema $\mathcal{S}$ of $\mathcal{D}$ is composed of (1) a set of N tables $\mathcal{T} = \{t_1, t_2, \cdots, t_N\}$, (2) a set of columns $\mathcal{C} = \{c_1^1, \cdots, c_{n_1}^1, c_1^2, \cdots, c_{n_2}^2, \cdots, c_1^N, \cdots, c_{n_N}^N\}$ associated with the tables, where n_i is the number of columns in the i-th table, (3) a set of foreign key relations $\mathcal{F} = \{(c_k^i, c_h^j) | c_k^i, c_h^j \in \mathcal{C}\}$, where

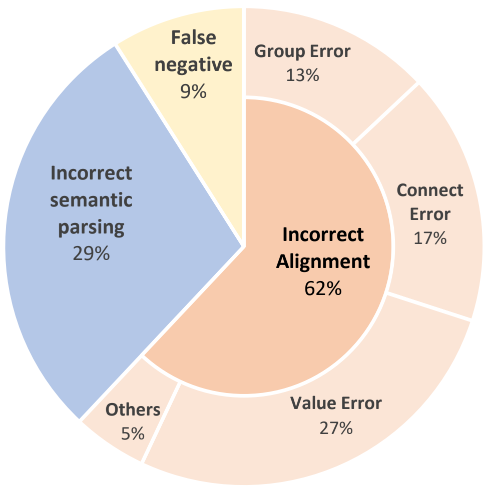


Figure 2: Statistic of simple zero-shot errors using GPT3.5-turbo

each (c_k^i, c_h^j) denotes a foreign key relation between column c_k^i and column c_h^j, (4) a set of primary key colomns $\mathcal{P} = \{c_k^i | c_k^i \in \mathcal{C}\}$, where each c_k^i uniquely identifies rows of table t_i.

3.1.2 Text-to-SQL Task

Formally, given a question q in natural language on certain database $\mathcal{D}$ with its schema $\mathcal{S}$, the Text-to-SQL task aims to maximize the possibility of LLM $\mathcal{M}$ generating the correct SQL s as follows: $\sigma\max \quad P_{\mathcal{M}}(s|\sigma(q, \mathcal{S}, \mathcal{D}))$, where function $\sigma(\cdot, \cdot)$ decides representation for target question q, with the useful information from the schema $\mathcal{S}$ and the database $\mathcal{D}$. Besides, $\sigma(\cdot, \cdot)$ also can include information such as instruction statement, rule implication and foreign key.

3.2 Error Analysis

We conducted a comprehensive error analysis on the wrong SQLs generated by three typical LLMs, including GPT3.5-turbo, GPT4, and PaLM. We manually identified the reason of their errors and found similar conclusions. For simplification, we only report results on GPT3.5-turbo in Figure 2.

We divide the errors into two main categories, i.e., incorrect semantic parsing and incorrect alignment. Incorrect semantic parsing means that the generated SQL does not correctly capture the question semantics, such as using wrong column in the `SELECT` clause. Incorrect alignment means the SQL captures the question semantics, but does not match the database schema as mentioned before. From Figure 2 we can see that incorrect alignment is the major reason for the errors, which accounts for 62%. Therefore, if the incorrect alignment problem is well addressed, it would greatly improve the performance of the task. We further divide the incorrect alignment errors into three major categories, i.e., the **Group Error**, the **Connect Error**, and the **Value Error**.

3.2.1 Group Error

Group error means that the LLM uses wrong column in the `GROUP` clause to identify an entity, which accounts for 13% of the incorrect alignment.

For example, given the question "Show the names of students who have likes, and numbers of pets for each", the LLM tends to use "`student.name`" (from the `SELECT` clause) in the `GROUP` clause to identify a student, which is incorrect in most cases if "`student.name`" is not the primary key ("`student.stuid`").

3.2.2 Connect Error

Connect error means that the LLM confuses the usage of logical connectors, i.e., `AND` or `INTERSECT`, `OR` or `UNION`, and `NOT IN` or `EXCEPT`. Connect error accounts for 17% of the incorrect alignment.

Connect error only happens when the question contains multiple conditions for the same entity or different entities. One typical example of connect error is shown in Figure 1. Typically, the LLM tends to use `AND`, `OR` and `NOT IN` to generate the SQL rather than `INTERSECT`, `UNION` and `EXCEPT` whose usage are more complex.

3.2.3 Value Error

Value error occurs when one entity has several identical descriptions, such as the "usa" and "the united state". Typically, the database content only contains one description, while the LLM may mistakenly use other descriptions that do not appear in the database in the condition clause if not given the database content. Value error acounts for 27% of the incorrect alignment.

The above three categories of errors cover almost all incorrect alignment. There are also some other incorrect alignment such as missing or redundant `DISTINCT` or `DESC`, which only account for a minor percentage. Besides, we also find some false negative samples, where the annotated gold SQLs are incorrect.

4 Our Method

To address above problems, we propose a novel **S**chema-**A**ligned framework (SA-SQL) to take full advantages of the database schema and correctly

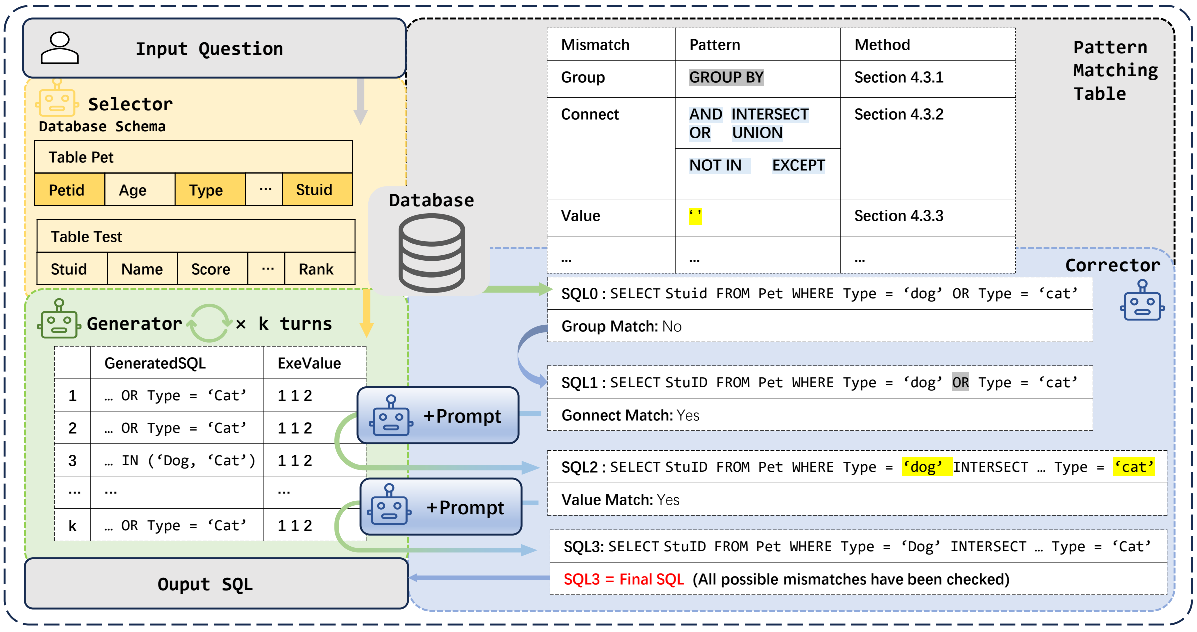


Table Pet				
Petid	Age	Type	···	Stuid

Table Test				
Stuid	Name	Score	···	Rank

	GeneratedSQL	ExeValue
1	… OR Type = ‘Cat’	1 1 2
2	… OR Type = ‘Cat’	1 1 2
3	… IN (‘Dog, ‘Cat’)	1 1 2
…	…	…
k	… OR Type = ‘Cat’	1 1 2

Mismatch	Pattern	Method
Group	GROUP BY	Section 4.3.1
Connect	AND INTERSECT OR UNION	Section 4.3.2
	NOT IN EXCEPT	
Value	‘ ’	Section 4.3.3
…	…	…

```
SQL0 : SELECT Stuid FROM Pet WHERE Type = ‘dog’ OR Type = ‘cat’
Group Match: No

SQL1 : SELECT StuID FROM Pet WHERE Type = ‘dog’ OR Type = ‘cat’
Gonnect Match: Yes

SQL2 : SELECT StuID FROM Pet WHERE Type = ‘dog’ INTERSECT … Type = ‘cat’
Value Match: Yes

SQL3: SELECT StuID FROM Pet WHERE Type = ‘Dog’ INTERSECT … Type = ‘Cat’
SQL3 = Final SQL (All possible mismatches have been checked)
```

Figure 3: The SA-SQL framework comprises three components: Selector, Generator, and Corrector

align the generated SQL with the schema. Considering that it would be pretty hard for the LLM to generate the complex SQL and check the possible incorrect alignment simultaneously, we design the framework following the draft-and-correction manner. As shown in Figure 3, the framework is composed of three key components, the *Selector*, the *Generator* and the *Corrector*. Specifically, the Selector first streamlines the database schema in accordance with user question by filtering irrelevant tables and columns, and thereby exclude the noise for generating a concise SQL and reduce input length. Next, the Generator generate the corresponding SQL in the zero-shot setting with self-consistency as a draft based on the schema. Last, the Corrector detects the parts that are prone to the incorrect alignment in the draft SQL based on a predefined Correction Table, and corrects the possible errors by regenerating the SQL with additional error-related information. In the following parts, we will introduce each component in detail.

4.1 Pattern Matching Table

As shown in Figure 3, we need to design regular expressions to extract corresponding error-prone SQL patterns. All SQLs that require grouping in their semantics will have the pattern `GROUP BY`. All SQLs with multiple conditions must use part of the connective patterns(e.g. `AND`, `INTERSECT`) and all SQLs that need to query the database text format content must use the pattern ’ ’. We actually completed the mapping from fuzzy semantics to definite features. We found that three qualitative features are enough to cover almost all misalignment problems. At the same time, since our prompt design allows the LLM to regenerate with sufficient knowledge rather than ordering it to correct errors, no matter how complex the matching features are, it will not cause misjudgment of the correct result of the initial inference.

4.2 Selector

The Selector is designed to automatically identify and select related tables and columns as prompts, which decreases the complexity of the database schema. The reason behind the process is that databases frequently comprise numerous tables and columns. The expansive nature of such schema may surpass the processing capacity of a LLM within one single call, which is a crucial problem particularly in real-world contexts. Consequently, the Selector is engineered to perform an initial filtration of relevant tables and columns to bolster efficiency. Moreover, most LLMs tend to use all inputs for generation, which may lead to a redundant or incorrect SQL. Excluding irrelevant information from the schema early could help generate a concise SQL.

To achieve the goal, we first select tables relevant to the input question by explicitly prompting the LLM to score the relevance between the question and each table based on the schema and taking the tables with higher scores. Then we select relevant columns from each relevant tables in a similar way. This strategy significantly enhances the efficacy of translating text into SQL, thereby improving both accuracy and efficiency.

4.3 Generator

The Generator aims to generate the SQL statement for the input question based on the selected tables and columns, serving as a draft for the Corrector.

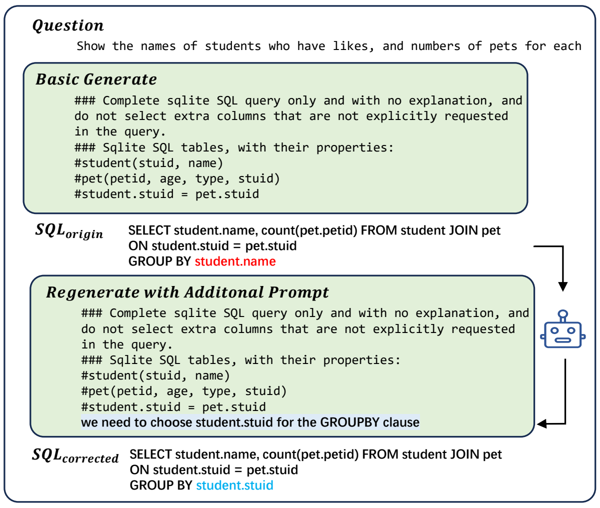
To maximize the use of the LLM's own reasoning capabilities, we use the OpenAI Demonstration Prompt, which is first used in OpenAI's official Text-to-SQL demo. It consists of instruction, table schemas, and question, where all information are commented by pound sign "#". Compared with other prompt format, the instruction in the OpenAI Demonstration Prompt is more specific with a rule, "Complete sqlite SQL query only and with no explanation", which is proven to significantly improve reasoning ability(Gao et al., 2023).

Moreover, to correctly generate the SQL statements, we also add foreign key information of the recalled tables in the context to specify the columns required for `JOIN` operations. Besides, for the primary key, the LLM could not well incorporate such information in SQL generation, so we use it in the Corrector for a full exploitation. Furthermore, we also apply the self-consistency technique(Wang et al., 2023) to improve the correctness of the generated SQL by sampling multiple different reasoning paths.

4.4 Corrector

Although the SQL draft by the Generator correctly captures the question semantics and conforms to the SQL syntax, there may exist errors in the alignment with the database schema as we have mentioned in Section 3.2. In other words, the correctness of the SQL may depend on the database schema, especially the primary key.

To this end, the Corrector detects the risky parts in the draft that are prone to alignment errors. Specifically, we define a Pattern Matching Table with SQL patterns for each risky part and corresponding fixing methods, with which the Corrector could check each alignment error one-by-one. Then the Corrector need to correct the possible errors in the risky parts. **It is worth noting** that most of the correction works provided vague and complex instructions to make the LLM to correct by itself, which is a one-way process and the effect is difficult to evaluate. Our method requires the Corrector to deduce the schema information which is crucial to align SQL in the current scenario, and feed it back to the Generator to regenerate SQL with self-consistency. This strategy makes our work a heuristic process of "educating the LLM to correct", instead of a prompt engineering of "ordering the LLM to find and correct", which better utilizes the powerful reasoning ability of the LLM itself.



```
Show the names of students who have likes, and numbers of pets for each
```

```
### Complete sqlite SQL query only and with no explanation, and
do not select extra columns that are not explicitly requested
in the query.
### Sqlite SQL tables, with their properties:
#student(stuid, name)
#pet(petid, age, type, stuid)
#student.stuid = pet.stuid
```

```
SELECT student.name, count(pet.petid) FROM student JOIN pet
ON student.stuid = pet.stuid
GROUP BY student.name
```

```
### Complete sqlite SQL query only and with no explanation, and
do not select extra columns that are not explicitly requested
in the query.
### Sqlite SQL tables, with their properties:
#student(stuid, name)
#pet(petid, age, type, stuid)
#student.stuid = pet.stuid
we need to choose student.stuid for the GROUPBY clause
```

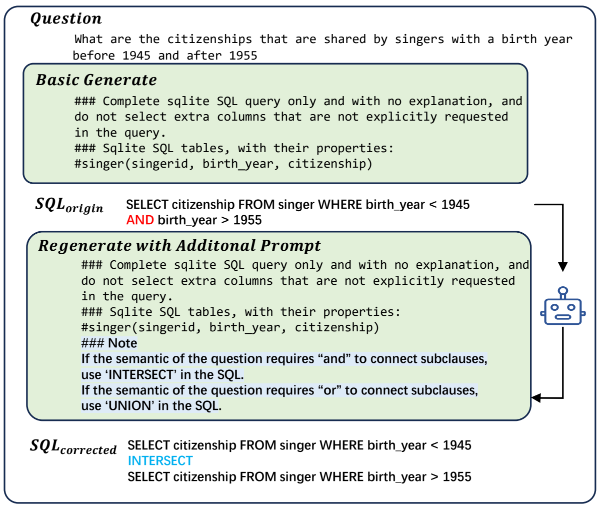
```
SELECT student.name, count(pet.petid) FROM student JOIN pet
ON student.stuid = pet.stuid
GROUP BY student.stuid
```

Figure 4: An example of Group Error and its correction process with addtional prompt provided by the Group Corrector

4.4.1 Group Corrector

As shown in Figure 4, the group error happens when the SQL contains the "`GROUP BY`" pattern. In these cases, the LLM tends to use the column (`student.name`) mentioned in the question to identify each entity to group by each student in the `GROUP` clause. If the column is the primary key, the SQL is correct in general; otherwise, there may exist two candidates of entity to group by, i.e., the column to select (`student.name` means grouping by each name), and the primary key (`student.stuid` means grouping by each student), which leads to a risk of error.

To fix the possible errors, we design a novel CoT prompt(Wei et al., 2022) Specifically, we first explicitly ask the LLM to recognize the entity to group from the two candidates based on the question semantics and then link the entity to the corresponding column in the table. Finally, we use this column to modify the column in the `GROUP` clause in the draft SQL, which will be extracted as the Corrector's output to augment the regeneration prompt.



Question

```
What are the citizenships that are shared by singers with a birth year before 1945 and after 1955
```

Basic Generate

```
### Complete sqlite SQL query only and with no explanation, and do not select extra columns that are not explicitly requested in the query.
### Sqlite SQL tables, with their properties:
#singer(singerid, birth_year, citizenship)
```

SQL_{origin} SELECT citizenship FROM singer WHERE birth_year < 1945 AND birth_year > 1955

Regenerate with Additonal Prompt

```
### Complete sqlite SQL query only and with no explanation, and do not select extra columns that are not explicitly requested in the query.
### Sqlite SQL tables, with their properties:
#singer(singerid, birth_year, citizenship)
### Note
If the semantic of the question requires “and” to connect subclauses, use ‘INTERSECT’ in the SQL.
If the semantic of the question requires “or” to connect subclauses, use ‘UNION’ in the SQL.
```

$SQL_{corrected}$ SELECT citizenship FROM singer WHERE birth_year < 1945
INTERSECT
SELECT citizenship FROM singer WHERE birth_year > 1955

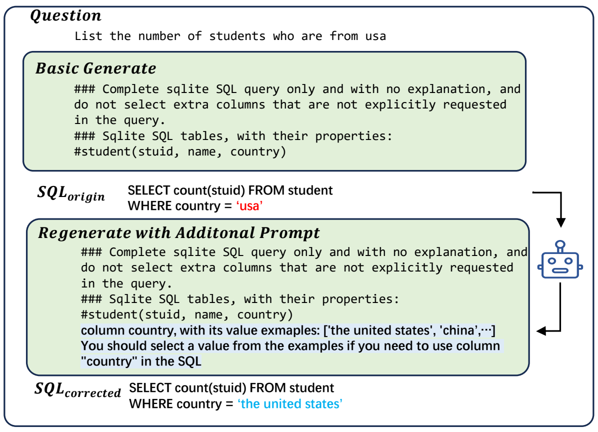
Figure 5: An example of Connect Error and its correction process with addtional prompt provided by the Gonnect Corrector

4.4.2 Connect Corrector

As shown in Figure 5, the connect error happens when there are multiple conditions in the SQL, featured by the `AND`, `OR`, `NOT IN`, and `INTERSECT`,`UNION`, `EXCEPT` patterns. These logical connectors have similar semantics, and thus the LLM only tends to use the structure represented by the former no matter what the scene is, even forcing the use of this structure comes at the expense of semantic changes.

Taking `AND` and `INTERSECT` as example, to clarify the differences, we assume there is an entity E in the SELECT clause and two attributes A1 and A2 describing it. In natural lanuage, "E (WHERE A1 AND A2)" and "E (WHERE A1) INTERSECT E (WHERE A2)" are completely equivalent. However, there are subtle semantic differences between the two in the SQL query. The reason is that in the SQL statement, WHERE subclause modifies the rows in the table rather than a single entity. For each row, there is a unique primary key corresponding to it. This is semantically equivalent to where describing the entity P corresponding to the primary key, rather than the entity E we selected in `SELECT`. If there is only one condition, there is no difference between the two. However, if there are multiple conditions modifying an object that is not a primary key, we can only split each condition out, and then modify it with `INTERSECT`, `UNION`, and `EXCEPT`, which represent logical `AND`, `OR`, and `NOT` respectively.

Through the CoT method, we enable the LLM to recognize the target entity modified by the original question clause and determine whether it corresponds to the primary key in the schema. It may be the selected object of `SELECT`, or it may be the parent node of the selected object. But unlike the Group Corrector, next we need to give the Generator different shots based on the judgment of whether the target entity corresponds the primary key and different logical relationships, and then let it regenerate SQL in the appendix. We show three different semantic corresponding GOLD SQL in the appendix, and give specific examples and explanations for each case.



Question

```
List the number of students who are from usa
```

Basic Generate

```
### Complete sqlite SQL query only and with no explanation, and do not select extra columns that are not explicitly requested in the query.
### Sqlite SQL tables, with their properties:
#student(stuid, name, country)
```

SQL_{origin} SELECT count(stuid) FROM student WHERE country = 'usa'

Regenerate with Additonal Prompt

```
### Complete sqlite SQL query only and with no explanation, and do not select extra columns that are not explicitly requested in the query.
### Sqlite SQL tables, with their properties:
#student(stuid, name, country)
column country, with its value exmaples: ['the united states', 'china',⋯]
You should select a value from the examples if you need to use column "country" in the SQL
```

$SQL_{corrected}$ SELECT count(stuid) FROM student WHERE country = 'the united states'

Figure 6: An example of Value Error and its correction process with addtional prompt provided by the Value Corrector

4.4.3 Value Corrector

As shown in Figure 6, the value error may happen when there is string matching in the condition of the SQL, featured by “‘” and “’”.

This type of error mainly comes from the misalignment of the database content and the natural language question. For example, the SQL draft follows the semantic "`SELECT count(studentP WHERE country = usa`" and will be executed on the database. If the database content is not completely consistent with the format mentioned in the question, such as capitalization, punctuation issues, etc., it will seriously lead to error. Through the CoT method, for the first generated SQL draft, we extract the columns that need to be queried for database content, and search for the values in the corresponding columns. Then, we ask the LLM to rerank the order of values according to their relevance to the format mentioned in the question and selects the most relevant k(We set k =3) values. Finally, we generate prompt with values and an instruction for the Generator to select item from values if needed.

5 Experiment

5.1 Experimental Setup

Datasets The Spider(Yu et al., 2018) dataset is the most challenging benchmark for the cross-domain and multi-table Text-to-SQL task. Spider contains a training set with 7,000 samples, a dev set with 1,034 samples, and a hidden test set with 2,147 samples, encompassing 200 distinct databases and 138 domains. In this study, we assess the efficacy of our framework on the Spider development set, as the test set is not accessible.

The BIRD (Li et al., 2023) dataset is a new benchmark for large-scale real databases, containing 95 large-scale databases and high-quality Text-SQL pairs, with a data storage volume of up to 33.4GB spanning 37 professional domains. Unlike Spider, BIRD focuses on massive and real database content, external knowledge reasoning between natural language questions and database content, and new challenges in SQL efficiency when dealing with large databases.

Evaluation Metrics According to standard requirements, we consider exact match accuracy (EM), execution accuracy (EX) on the Spider and execution accuracy (EX), valid efficiency score (VES) on the BIRD to evaluate Text-to-SQL models.
EM treats each clause as a set and compares the prediction for each clause to its corresponding clause in the reference query. A predicted SQL query is considered correct only if all of its components match the ground truth. This metric does not take values into account.
EX is defined as the proportion of questions in the evaluation set for which the execution results of both the predicted and ground-truth inquiries are identical, relative to the overall number of queries.
VES is designed to measure the efficiency of valid SQLs generated by models. It is important to note that "valid SQLs" refers to predicted SQL queries whose result sets align with those of the ground-truth SQLs.

Baselines We conduct experiments on both BIRD and Spider dataset and compare our method with the following baseline:

- **ChatGPT+CoT** (Tai et al., 2023) uses simple zero-shot text-to-SQL prompt for SQL generation.

Approach	Spider Dev EM	Spider Dev EX	BIRD Dev VES	BIRD Dev EX
Pre-trained model methods				
T5-3B	71.5	74.4	25.5	10.3
RESDSQL-3B + NatSQL	80.5	84.1	-	-
Graphix-3B + PICARD	77.1	84.1	-	-
Large language model methods				
GhatGPT	48.8	75.5	43.8	37.2
ChatGPT+CoT	49.6	76.2	42.3	36.6
C3-SQL	56.2	81.8	46.3	39.0
GPT-4	40.4	72.9	49.7	46.3
DIN-SQL+GPT4	60.1	82.8	58.7	50.7
DAIL-SQL+GPT4	68.7	83.6	56.8	54.7
Our proposed method				
Our Method	**69.2**	**89.8**	60.3	59.3

Table 1: EM and EX results on Spider's development set and BIRD's development set (%). We compare our approach with some powerful baseline methods from the top of the official leaderboard of Spider and BIRD

- **C3-SQL** (Dong et al., 2023) first performs schema linking filtering and then directs GPT3.5-turbo with a calibration bias prompt designed for Spider with a self-consistency strategy.

- **DIN-SQL** (Pourreza and Rafiei, 2023) decomposes the text-to-SQL task into smaller subtasks and designs different prompts for each subtask to instruct GPT-4 to complete each subtask and obtain the final SQL.

- **DAIL-SQL** (Gao et al., 2023) encodes structure knowledge as SQL statements, selects few-shot demonstrations based on their skeleton similarities.

5.2 Result

As shown in Table 1, for the development set of Spider, our method achieves the highest execution accuracy and execution accuray, and outperforms the 2rd method by 0.5% and 6.2%, which proves the effectiveness of our approach in solving schema-alignment problem.

For the development of BIRD, although we surpassed all methods using GPT4 as backbone, we are still far behind the first place on the leaderboard which is anonymous. The additional challenge of BIRD lies in effectively leveraging the specific domain knowledge offered by the data set, the top-1 and top-2 solutions on the leaderboard have a SFT procedure, while SA-SQL has no such procedure

	EM	EX
SA-SQL	69.2	89.8
w/o Selector	64.0	87.6
w/o Corrector	59.8	83.5
w/o Group Corrector	60.8	88.5
w/o Connect Corrector	68.2	88.7
w/o Value Corrector	69.2	85.5

Table 2: SA-SQL ablation study in Spider dev set. SA-SQL using GPT-4 by default.

Error Category	**GPT3.5-turbo**		
	Resolved	Unresolved	All
Incorrect Schema Parsing	4	25	29
Incorrect Alignment	49	13	62
Group	13	0	13
Connect	10	7	17
Value	26	0	26
Others	1	7	8
False Negative	0	9	9

Table 3: Analysis of 100 sample errors made by GPT3.5-turbo with zero-shot. We group the errors into 3 categories and examine if our model resolves them.

and requires further design to incorporate extra knowledge for a more fair comparison.

5.3 Ablation Study

We perform the ablation study in Spider dev set using GPT-4. The results prove the effectiveness of every module we designed. The improvement of EM is mainly provided by the Group Corrector, which is mainly caused by the model's misjudgment of `count(*)` and `count(the colomn responding *)`. The improvement of EX is mainly provided by the Value Corrector, which is mainly caused by the model's misalignment with database value.

5.4 Error Analyse

We also study the effectiveness of the Corrector on the alignment errors we found, and report the results in Table 3. As shown in the Table, the incorrect alignment accounts for 62% of all errors, and our method has corrected most of the alignment errors. Specifically, our method has corrected all group errors and value errors, which account for a major part of the incorrect alignment. For the connect error which is more complex, our method also correctly fix more than half of the errors. The results have proved the effectiveness of our Corrector module.

6 Conclusion

Prompting has enabled large language models to achieve impressive performance on numerous NLP tasks across different domains, without requiring a large training set. However, the performance of prompting approaches for the Text-to-SQL task can not make great progress. In this study, we have defined the schema-alignment subtask to address Text-to-SQL task using prompting. Our experimental results demonstrate that our method can effectively bridge the gap between natural lanuage and SQL query, which will make the LLM reliable database interface.

Limitations

In the Corrector of our proposed SA-SQL framework, whether information enhancement is performed on SQL depends on the results of manually defined pattern matching. This means that our framework can only find defined error types. Although our definition is enough to cover the vast majority of errors, and the framework also supports adding new pattern matching rules, the overall work is still not elegant enough. In the future work, we will attempt to fine-tune entity a Text2SQL LLM with high quality error correction samples to improve the integrity of the framework and its ability to correct undefined errors.

Another limitation is that the framework may need to interact with LLMs multiple times when facing complex SQL, introducing end-to-end latency. This may have an impact on the practical application of the framework. In response to this, we can consider replacing online closed-source LLMs with locally deployed LLMs to reduce latency.

References

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA. Curran Associates Inc.

Naihao Deng, Yulong Chen, and Yue Zhang. 2022. Recent advances in text-to-SQL: A survey of what we have and what we expect. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 2166–2187, Gyeongju, Republic of Korea. International Committee on Computational Linguistics.

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.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. Text-to-sql empowered by large language models: A benchmark evaluation. *CoRR*, abs/2308.15363.

Jinhao Jiang, Kun Zhou, Zican Dong, Keming Ye, Xin Zhao, and Ji-Rong Wen. 2023. StructGPT: A general framework for large language model to reason over structured data. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 9237–9251, Singapore. Association for Computational Linguistics.

Jinyang Li, Binyuan Hui, GE QU, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin 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 *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Aiwei Liu, Xuming Hu, Lijie Wen, and Philip S. Yu. 2023. A comprehensive evaluation of chatgpt's zero-shot text-to-sql capability.

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 the Association for Computational Linguistics: EMNLP 2023*, pages 14935–14956, Singapore. Association for Computational Linguistics.

Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Ruoxi Sun, Sercan Arik, Rajarishi Sinha, Hootan Nakhost, Hanjun Dai, Pengcheng Yin, and Tomas Pfister. 2023. SQLPrompt: In-context text-to-SQL with minimal labeled data. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 542–550, Singapore. Association for Computational Linguistics.

Chang-Yu Tai, Ziru Chen, Tianshu Zhang, Xiang Deng, and Huan Sun. 2023. Exploring chain of thought style prompting for text-to-SQL. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 5376–5393, Singapore. Association for Computational Linguistics.

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 *The Eleventh International Conference on Learning Representations*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Appendices

We provide more details omitted in the main text as follows:

- Appendix A: Implementation Details
- Appendix B: Detail in the Selector
- Appendix C: Detail in the Generator
- Appendix D: Detail in the Corrector

A Implementation Details

All our modules are based on GPT4 given its excellent inherent reasoning capabilities. We determine the temperature and whether to use in-context learning based on the functional characteristics of the respective modules. In order to be applicable to practical applications in the future, we only interact with LLM and consume tokens in the necessary components. Generally,

- the **Seclector** was based on GPT4 with temperature = 0.7 and zero-shot setting. At the same time, we select top k1 = 3 tables and top k2 = 10 columns in each top tables.

- the **Generator** was based on GPT4 with temperature = 0.7 and zero-shot setting. We applied self-consistency strategy with k = 20, which means that the generated SQL was the SQL with the highest frequency of execution results among 20 independent generations.

- the **Connector** was composed of rule-based pattern matching and prompt-based knowledge enhancement. For the pattern matching, it is worth noting that we only set up 3 rounds of detection because this is enough to cover the vast majority of possible error scenarios in SQL generation, not just these three error scenarios. An important design principle of the framework is to ensure the independence of each correction module. Readers can modify, delete or add new matching rules and corresponding enhanced knowledge at will without affecting the normal use of other modules. For the knowledge enhancement, each round only needs to perform one turn of interaction with LLM.
  Group Corrector was based on GPT4 with temperature = 0 and 5-shot setting.
  Connect Corrector was based on GPT4 with temperaturen = 0 and 5-shot setting.
  Value Corrector was based on GPT4 with temperature = 0.7 and zero-shot setting.

B Detail in the Selector

As shown in the Figure D.1, we need to do two rounds of selection. In the first round we exclude completely irrelevant tables, and in the second round we exclude irrelevant columns in the selected table.

C Detail in the Generator

As shown in the Figure D.2, we applied self-consistency strategy with k = 20, which means that the generated SQL was the SQL with the highest frequency of execution results among 20 independent generations. The setting is due to the following reason: our paper is based on the assumption that the initially generated SQL draft has completely extracted the semantic information of natural language, and errors only occur during alignment. Therefore, in this step, we must maximize the reasoning ability of LLM itself as much as possible.

D Detail in the Corrector

Each of our sub-correction modules follows the following paradigm design:

1. Match the designed pattern to SQL
2. Find the corresponding information in the database and evaluate whether information enhancement is needed
3. Use LLM to analyze new information and original SQL to draw corresponding conclusion
4. Design additional prompts based on the conclusion and regenerate SQL

To minimize token overhead, only the regeneration in the fourth step and the third step are based on LLM, while the other steps are based on rule.

D.1 Group Corrector

The key to this module is to let the LLM reason about the column used for group based on CoT without noise. We follow the paradigm:

1. Match the pattern **GROUP BY** with SQL
2. If matched, extract the intersection of column in the `SELECT` and the primary key of its table. If there are more than one item in the intersection, SQL generation may encounter difficulty when selecting and information enhancement is required. Otherwise, exit the Group Corrector.
3. Input the intersection of columns as an alternative set and the original problem into LLM, perform as the Figure D.3
4. Design additional prompt following the format and regenerate SQL. **Chosen_column** is the output of LLM in Step 3
   {we need to choose **Chosen_column** for the GROUPBY clause}

D.2 Connect Corrector

The key to this module is to let the LLM reason about the true entity modified by sub-clause based on CoT without noise. For exmaple, in the clause "name of student who is 10 years old", sub-clause "who is 10 years old" decorates the entity "student" instead of entity "name". Moreover, there are two different formats based on whether the entity is the primart key in three contexts representing `AND`, `OR` and `NOT`. We follow the paradigm:

1. Match the patten **AND | INTERSECT | OR | UNION | NOT IN | EXCEPT** with SQL
2. If matched, extract the intersection of column in the `SELECT` and the primary key of its table. If there are more than one item in the

intersection, SQL generation may encounter difficulty when selecting and information enhancement is required. Otherwise, exit the Connect Corrector.

3. Input the intersection of columns as an alternative set and the original problem into LLM, perform as the Figure D.4

4. Judge whether the **Chosen_column** is the primary key and the context type. There are 6 types of additional prompts. **Chosen_column** is the output of LLM in Step 3. Let's take "the column is not the primary key" and "AND" context as an example:
{If the semantic of the question requires "and" to connect subclauses,use 'INTERSECT' in the SQL. }

D.3 Value Corrector

The key to this module is to let the LLM filter out values relevant to the question and use item from them in the next generation.

1. Match the pattern **' '** with SQL

2. If matched, extract the column which contains the pattern and find its distinct values in the database. SQL generation may encounter wrong prediction in the first generation because LLM can only obtain information from question at that time.

3. Input the distinct values as an alternative set and the original problem into LLM, perform as the Figure D.5

4. Design additional prompt following the format and regenerate SQL. **Value_examples** is the output of LLM in Step 3 , and **column** is extracted in Step 2
{**Value_examples**
You should select a value from the examples if you need to use **column** }

D.4 More Corrector

Our framework is modular which means readers can modify, delete or add new matching rules and corresponding enhanced knowledge at will without affecting the normal use of other modules. Even if the new rule matching object duplicates the original rule, "regenerate rather than correcting" manner can ensure that the framework will not change the original correct SQL into an incorrect one.

Table Recall Prompt Template:

Given the database schema and question, perform the following actions:
1 - Rank all the tables based on the possibility of being used in the SQL according to the question from the most relevant to the least relevant, Table or its column that matches more with the question words is highly relevant and must be placed ahead.
2 - Check whether you consider all the tables.
3 - Output a JSON object that keeps the top 3 tables.
### Schema:{schema}
### Foreign_ keys:{foreign_ keys}
### Question:{question}
:

Column Recall Prompt Template:

Given the database tables and question, perform the following actions:
1 - Rank all the columns in each table based on the possibility of being used in the SQL, you should output them in the order of the most relevant to the least relevant.
2 - Check whether you consider all the colums.
3 - Output a JSON object that keeps the top 10 columns in each table.
### Schema:{schema}
### Foreign_ keys:{foreign_ keys}
### Question:{question}
:

Table D.1: Prompt template in the Selector

Basic Generator Template:

### Complete sqlite SQL query only and with no explanation, and do not select extra columns that are not explicitly requested in the query.
### Sqlite SQL tables, with their properties:
#
#{schema}
#
# Foreign_ keys:{foreign_ keys}
### {question}
SELECT

Table D.2: Prompt template in the Generator

Group Corrector Template:

[System]:
Given a question, you should use the provided alternative columns to complete the SQLite SQL QUERY for the GROUP BY clause.
Note: you must choose only one column in the alternative columns which is most relavant to the entity in the question which you need group.

Shot * 5:
[User]:
Table stadium, with the alternative columns[stadium_id(Primary_key), name]
Question: Show the stadium name and the number of concerts in each stadium
[Assistant]:
Answer: Let's think step by step. To answer the question "Show the stadium name and the number of concerts in each stadium", we need group stadium and select the number of concerts in each stadium. So in the SQLite SQL Query for the given question, we need to choose **stadium.stadium_id**, which is most relevant to stadium for the GROUPBY clause
Chosen column: stadium.stadium_id
...

[User]:
{primary table and alternative columns}
Question: {question}
[Assistant]:

Table D.3: Group Corrector: temperature = 0 and 5-shot

Connect Corrector Template:

[System]:
Given a question, you should follow my instruction to choose the most relevant entity.
Step 1, you should find which is the answer to the question and its corresponding sub-clause. The target entity which sub-clause describes should be in the noun phrase.
Step 2, you should choose only one column in the alternative columns for the following SQL generation. The column should be most relevant to the target entity in the alternative columns.
Note: you must choose only one column!

Shot * 5:
[User]:
Table singer, with the alternative columns[id(Primary_key), citizenship]
Question: List the citizenship of singers born in 2014.
[Assistant]:
Answer:
Let's think step by step. To answer the question "List the citizenship of singers born in 2014", we need to find the answer for "citizenship of singers". The corresponding sub-clause is "born in 2014", which describes singers.
Target entity: singers
So we need to choose singer.id, which is primary key of singer and most relevant to the target entity "singers".
Chosen column: singer.id
...

[User]:
{primary table and alternative columns}
Question: {question}
[Assistant]:

Table D.4: Connect Corrector: temperature = 0 and 5-shot

Value Corrector Template:

Given the alternative values and question, perform the following actions:
1 - Rank all the values based on the possibility of being used in the SQL according to the question from the most relevant to the least relevant, Values that matches more with the question words is highly relevant and must be placed ahead.
2 - Check whether you consider all the values.
3 - Output a JSON object that keeps the top 3 values.
### {columns and its alternative values}
### Question: {question}

Table D.5: Value Corrector: temperature = 0.7 and zero-shot