

OpenSearch-SQL: Enhancing Text-to-SQL with Dynamic Few-shot and Consistency Alignment

Xiangjin Xie

xiexiangjin.xxj@alibaba-inc.com
Alibaba Cloud
China

LingYan Zhao

zhaolingyan.zly@alibaba-inc.com
Alibaba Cloud
China

ABSTRACT

Although multi-agent collaborative Large Language Models (LLMs) have achieved significant breakthroughs in the Text-to-SQL task, their performance is still constrained by various factors. These factors include the incompleteness of the framework, failure to follow instructions, and model hallucination problems. To address these problems, we propose OpenSearch-SQL, which divides the Text-to-SQL task into four main modules: Preprocessing, Extraction, Generation, and Refinement, along with an Alignment module based on a consistency alignment mechanism. This architecture aligns the inputs and outputs of agents through the Alignment module, reducing failures in instruction following and hallucination. Additionally, we designed an intermediate language called SQL-Like and optimized the structured CoT based on SQL-Like. Meanwhile, we developed a dynamic few-shot strategy in the form of self-taught Query-CoT-SQL. These methods have significantly improved the performance of LLMs in the Text-to-SQL task.

In terms of model selection, we directly applied the base LLMs without any post-training, thereby simplifying the task chain and enhancing the framework's portability. Experimental results show that OpenSearch-SQL achieves an execution accuracy(Ex) of 69.3% on the BIRD development set, 72.28% on the test set, and a reward-based validity efficiency score (R-VES) of 69.36%, with all three metrics ranking first at the time of submission. These results demonstrate the comprehensive advantages of the proposed method in both effectiveness and efficiency.

KEYWORDS

Text-to-SQL, Agent, Large Language Model

ACM Reference Format:

Xiangjin Xie, Guangwei Xu, LingYan Zhao, and Ruijie Guo. 2018. OpenSearch-SQL: Enhancing Text-to-SQL with Dynamic Few-shot and Consistency Alignment. In *Proceedings of Make sure to enter the correct conference title*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06...\$15.00
<https://doi.org/XXXXXXX.XXXXXXX>

Guangwei Xu

kunka.xgw@alibaba-inc.com
Alibaba Cloud
China

Ruijie Guo

ruijie.guo@alibaba-inc.com
Alibaba Cloud
China

from your rights confirmation email (Conference acronym 'XX). ACM, New York, NY, USA, 15 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 INTRODUCTION

Text-to-SQL task attempts to automatically generate Structured Query Language (SQL) queries from Natural Language Queries (NLQ). This task can improve the ability to access databases without the need for knowledge of SQL [17]. As the text-to-SQL problem is notoriously hard, these systems have been the holy grail of the database community for several decades [17]. Early work defined query answers as graph structures [14, 15] or based on syntactic structures for parsing the questions [16, 38]. Subsequent approaches have treated the Text-to-SQL task as a neural machine translation (NMT) problem [11, 37]. Recently, with the development of large language models, researchers have increasingly accomplished the task through methods such as the use of supervised fine-tuning (SFT) [19, 23, 35], Chain-of-Thought(CoT) [40], Agents [6, 27], and In-Context Learning [7, 36], achieving results that greatly surpass previous methods.

Limitation. Although the methods driven by LLMs have significantly raised the upper limits of the Text-to-SQL task capabilities, our analysis of previous work reveals that:

L1 Due to the ambiguity of the overarching framework, there are some gaps at the methodological level. This has prevented these methods from reaching their potential. For example, there is a lack of verification of the stored information in the database, no error correction for the generated results, and issues related to the absence of few-shot learning.

L2 LLM-driven approaches often rely on multi-agent collaboration. However, due to the instability of LLMs and the lack of guaranteed coherence and coupling between agents, later-executing agents may not use or only partially use the outputs of previously running agents. This leads to accumulated hallucinations and performance loss.

L3 The instructions and steps that guide the LLMs significantly affect the quality of the generated SQLs. In the pre-LLM era, methods employing intermediate languages to generate SQL were developed to address this issue [17], but current research on instruction construction remains insufficient.

Inspired by the aforementioned challenges, we conducted a detailed analysis of the human workflow in completing the Text-to-SQL task: understanding the database structure and selecting the

specific tables, columns, and values needed to get the SQL. This process often requires executing several simple SQL queries, such as investigating the specific forms of values for a certain column or retrieving how a particular value is stored in the database. Next, based on the specifics of the NLQ, the appropriate aggregation functions and SQL syntax are chosen to construct the backbone of the SQL query, followed by filling in the relevant statements. Finally, humans typically modify the SQL incrementally based on the query results until the requirements are met. For more complex questions, humans often choose to consult or borrow from others' formulations to seek solutions. Based on this approach, we developed the OpenSearch-SQL method, with detailed as follows.

Framework. To address **L1**, our research defines a **Standard Text-to-SQL framework** based on the process humans use to complete SQL construction. This can cover current research on LLM-driven Text-to-SQL tasks. We believe that a complete Text-to-SQL task framework should include four steps: **Preprocessing, Extraction, Generation, Refine:**

- **Preprocessing.** Processing and constructing all auxiliary information that is insensitive to the NLQs, including database schema information, embedding databases, few-shot libraries, etc.
- **Extraction.** Extract and select the necessary elements for generating SQL, including sub NLQ, few-shot, as well as tables, columns, values, etc., from the database.
- **Generation.** Translate NLQ into SQL based on the SQL syntax, few-shot and the prepared information.
- **Refine.** Check and optimize the SQL based on the execution results using alignment strategies and rules. Then, select the final SQL based on self-consistency and voting results.

For **L2**, we analyzed common hallucinations in multi-agent collaboration-driven Text-to-SQL tasks and designed a consistency alignment-based **alignment agent** for these hallucinations. By ensuring consistency in the functions, inputs, and outputs of specific agents, our aim is to reduce hallucinations during the model generation process. This alignment approach can also be extended to make targeted adjustments to the agents' outputs. As shown in Figure 1, we proposed the OpenSearch-SQL framework, which consists of a standard Text-to-SQL framework combined with alignments.

Furthermore, we extended our research on multi-agent methods driven by LLMs using this framework. For **L3**, we developed a self-taught few-shot augmentation mechanism that supplements query-SQL pairs with **Chain-of-Thought(CoT)** information, creating **Query-CoT-SQL** pairs. This enriched few-shot approach can enhance the performance of LLMs. Additionally, we designed an intermediate language named SQL-Like within the CoT logic, allowing LLMs to first generate the backbone of the SQL before the specific SQL details, thus reducing the complexity of the task for the models.

In real-world applications, preparing a large amount of customized SQL training data for specific query tasks is not an easy task. Therefore, considering the needs of the real world, our goal is not only to achieve good results in specific evaluation tasks, but also to ensure that the proposed method has strong transferability. We aim for it to be easily applicable to various database query tasks without the need for post-training. This means that when faced

with any issue, we can simply modify the agent and alignment methods without the need to retrain the model, thus meeting user needs with high quality.

Based on this, we directly implemented our approach on a pre-trained LLM model without employing any supervised fine-tuning (SFT) or reinforcement learning (RL). In terms of experiments, we chose the widely recognized BIRD dataset [20] for testing, which records three metrics: Execution Accuracy(EX) on the development set, EX on the test set, and the Reward-based Valid Efficiency Score(R-VES) on the test set. Our experimental results indicate that we achieved 69. 3% EX on the Dev set, 72. 28% EX on the Test set, and an R-VES score of 69.3%. All three metrics ranked first on the leaderboard at the time of submission, demonstrating that our method not only generates accurate SQL statements but also has a clear advantage in terms of time efficiency. We hope that this consistency alignment-based approach will inspire future research to explore more efficient consistency alignment strategies.

Contribution In summary, our contributions are as follows.

- To the best of our knowledge, we are the first to propose a multi-agent cooperative framework based on consistency alignment for the Text-to-SQL task. This alignment mechanism significantly reduces the loss of information transmission between Agents and the hallucinations in the results generated by the Agents.
- We introduce a novel self-taught dynamic few-shot method and a SQL-Like CoT mechanism. This enhances the performance and stability of LLMs when generating SQL.
- Our proposed method, OpenSearch-SQL, did not utilize fine-tuning or introduce any specialized datasets. At the time of submission, it achieved 69.3% EX on the BIRD benchmark validation set, 72.28% EX on the test set, and R-VES of 69.36%, all ranking first.

2 PRELIMINARY

We first introduce the two important modules this paper relies on: Large Language Models (LLMs) and the Text-to-SQL task.

2.1 Large Language Models

Pre-trained large language models have acquired extensive language knowledge and semantic information through unsupervised learning on large-scale textual data. This pre-training typically employs autoregressive architectures (such as GPT-like models) or autoencoding structures (such as BERT-like models). The LLMs can achieve excellent performance on various downstream tasks through fine-tuning. Since the advent of the deep learning era for large language models, there have been significant advancements in Text-to-SQL tasks compared to classical models and methods, with current SOTA-level work fully relying on the performance of LLMs.

2.2 Text-to-SQL

The Text-to-SQL task can be defined as: a task that translates a NLQ Q into an SQL query Y based on database information S , a specific model $F(\cdot|\theta)$, and a particular prompt P . The information in the database is defined as S . Therefore, the Text-to-SQL task can

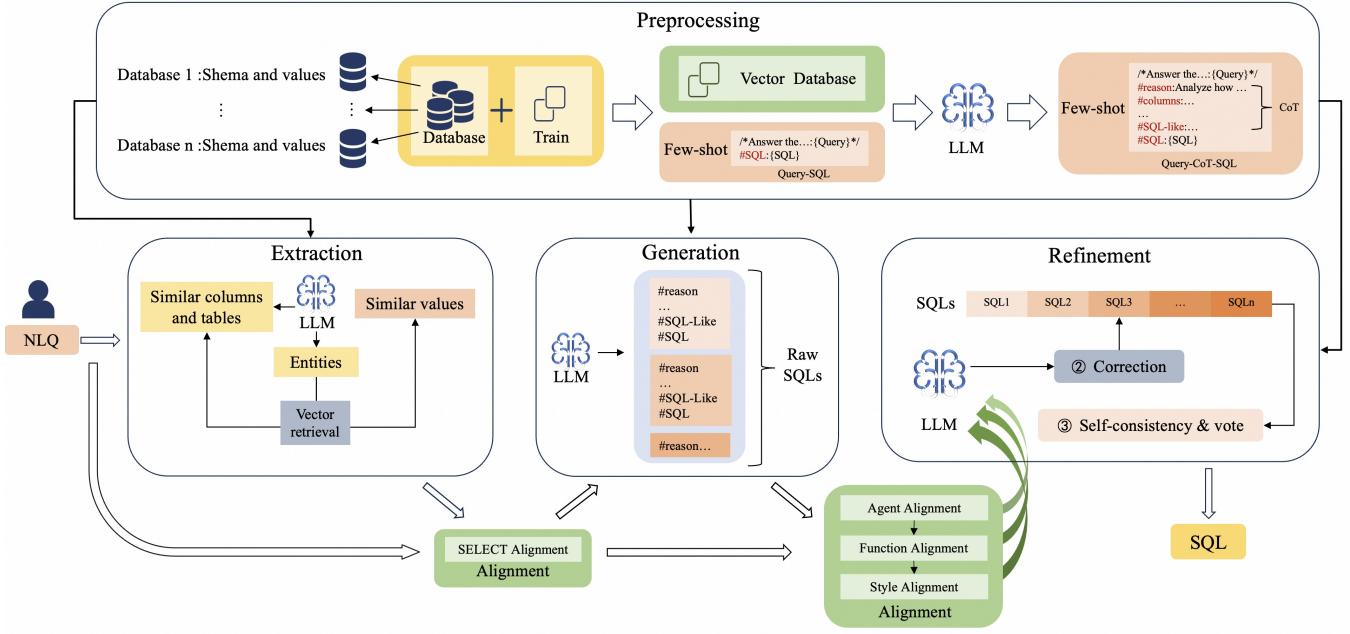


Figure 1: The basic framework of OpenSearch-SQL optimizes the Text-to-SQL task through a multi-agent collaboration approach based on consistency alignment.

be defined as:

$$Y = F(Q, S, P|\theta), \quad (1)$$

In the current state-of-the-art Text-to-SQL tasks, multi-agent collaboration has become a key strategy for enhancing performance. Common modules include value retrieval, schema linking, few-shot driving, CoT prompting, SQL correction, and self-consistency. Inspired by these methods, this paper first systematically organizes these modules. We categorize existing methods into four stages: **Preprocessing**, **Extraction**, **Generation**, and **Refinement**. The **Preprocessing** is independent of a specific NLQ and aims to obtain clear information about the database structure and prepare other auxiliary information that aids the model in generating SQL; the **Extraction** analyzes specific queries and filters out high-quality auxiliary information from the prepared information; the **Generation** organizes the extracted information and generates candidate SQLs; and the refinement stage further optimizes the candidate SQLs to enhance the quality of SQLs.

Based on this foundation, we propose OpenSearch-SQL, a Text-to-SQL framework based on **dynamic few-shot** and **consistency alignment** mechanisms. The design of dynamic few-shot aims to strengthen the performance of each module across the various stages of the task, while the **Alignment** module is used to align the different stages of Text-to-SQL, thereby reducing the hallucination produced by LLMs in multi-agent collaboration workflows. The design of OpenSearch-SQL simplifies the complex instruction system, focusing on improving multi-agent collaboration effectiveness through alignment. In the following sections, we will provide a detailed introduction to the specific components of the OpenSearch-SQL framework.

2.3 Hallucination

Hallucination [46] is a common problem in deep learning, usually referring to situations where the content generated by a model does not align with reality or expected outcomes, i.e., the model produces inaccurate or unreasonable information. Textual hallucinations can manifest in the following forms:

- **Irrelevant content:** The text generated by the model is unrelated to the input or context, or it includes irrelevant details.
- **Incorrect facts:** The generated text contains erroneous facts or information that do not exist or are inaccurate in the real world.
- **Logical errors:** The generated text is logically inconsistent or unreasonable.

In Text-to-SQL, this manifests as: results containing non-existent database information, failure to follow instructions in the prompt, and typographical and syntactic errors. Additionally, for convenience in handling, in the Text-to-SQL task, we also include the result deviations caused by randomness at non-zero temperatures and errors due to minor prompt changes. Because the essence of hallucination is the discrepancy between the training process and the actual use scenarios of the mode. Therefore, for hallucination problems, common handling methods include: post-training specifically for hallucination problems, using the Self-Consistency mechanism to reduce randomness, and post-processing. Since this paper focuses on optimizations at the architectural level, our work aims to reduce hallucinations through the last two methods mentioned above.

3 METHODOLOGY

In this section, we detail the specifics of OpenSearch-SQL. We provide a more in-depth description of the core concepts of Alignments and dynamic few-shot before outlining the entire process. To ensure a smooth explanation, we integrate each stage with its corresponding Alignments and describe them in the order they operate within the OpenSearch-SQL framework.

3.1 Alignments

LLM hallucination [46] is a critical problem affecting the usability of LLMs, and it is also present in the Text-to-SQL task. Furthermore, the cumulative errors that arise from multi-agent collaboration exacerbate the hallucination problem in LLMs. For example, if an agent responsible for the **Extraction** function selects columns from the database in a certain way and generates incorrect column names due to model hallucination, this error will persist in the subsequent **Generation**, as it lacks knowledge of the correct database structure. Consequently, the hallucination produced by the previous agent is inherited, making it difficult for hallucinations to spontaneously disappear in a multi-agent LLM workflow; the total amount of hallucination is almost monotonically non-decreasing.

Based on this phenomenon, we define the Alignment Agent as :

$$A_{Alignment}(x + A'(x)) = A(x) - A'(x), \quad (2)$$

The input for the agent is the input x for the agent to be aligned, along with $A'(x)$. The output is $A_{Alignment}(x + A'(x))$, which is obtained through optimization by a large model and rules. Empirical observations suggest that the difference between $A'(x)$ and $A(x)$ is mainly due to hallucinations in the model. Therefore, the function of Alignment is to align the current Agent's input information with its output result, so that this alignment can reduce the hallucinations between the expected output and the actual output.

In the Text-to-SQL task, common hallucination phenomena mainly refer to failures in instruction following and instability in output results. This is specifically manifested as: generating non-existent columns, changing database column names, syntax errors, mismatching database values with columns, and failing to adhere to the rules set in the prompt. To address these problems, we propose a consistency alignment method: after each agent completes its output, an **Alignment** agent is used to align the current agent's output with that of the upstream agent, ensuring that the functionalities of the various agents achieve logical consistency, and the aligned result is passed on to the downstream agent. This mechanism is somewhat analogous to residual connections [13], effectively extending the chain of collaboration among multiple agents while minimizing the introduction of hallucinations. The details of each Alignment will be presented in detail in the following Agent introduction.

3.2 Self-Taught Fewshot

Few-shot is an important method to assist LLMs in generation, MCS-SQL[18], DAIL-SQL[7] studied the critical role of problem representation in the Text-to-SQL task and proposed using question similarity to select appropriate few-shots to drive LLMs in generating SQL, achieving notable results.

Inspired by this, we attempted to use dynamic few-shots to enhance the efficiency of agents in the Text-to-SQL task. Additionally, we considered how to better leverage few-shots by extracting more information from the samples. Therefore, in OpenSearch-SQL, we first used **Masked Question Similarity (MQs)** [10] to select similar queries. Then, we upgraded the few-shots in the Query-SQL format by self-taught. For the Query-SQL pairs shown in Listing 1, we used an LLM to supplement the CoT information to transform the NLQ into SQL. As shown in Listing 2, this resulted in **Query-CoT-SQL** pairs containing logical information. Compared to simple Query-SQL pairs, these self-taught few-shots provide richer information.

Then, for the error correction of the **Refinement**, we prepared different few-shots for various error types, enabling LLMs to more clearly understand how to correct SQL based on different errors during **Refinement**. The format of the few-shots is shown in Listing 3, the specific errors in the **Raw SQL** and the **Advice of correction** correspond to the error type.

Listing 1: Format of Few-shot in Query-SQL Pair

```
/* Answer the following:{question} */
#SQL: {SQL}
```

Listing 2: Format of Few-shot in Query-CoT-SQL Pair

```
/* Answer the following:{question} */
#reason: Analyze how to generate SQL based
on the question.
#columns: All columns ultimately used in SQL
#values: the filter in SQL
#SELECT: SELECT content table.column.
#SQL-like: SQL-like statements ignoring Join
conditions
#SQL: {SQL}
```

Listing 3: Format of Few-shot in Correction

```
{"Result: None": "/* Fix the SQL and
answer the question */",
"question: {question}",
"Error SQL: {Raw SQL}",
"Error: Result: None",
"values: {values in Database}",
"Change Ambiguity: {Advice of correction}",
"SQL:{corrected SQL}""",
... }
```

3.3 Preprocessing

In the **Preprocessing**, we constructed the database based on its true structure. Additionally, to ensure that the SQL generated by LLMs aligns with the actual state of the database, we indexed the values within the database. This allows the SQL to avoid errors that might arise from small character discrepancies. It's worth noting that we index only string-type data to save the space required for building the retrieval database.

In addition, the construction of dynamic Fewshot has also been completed in this phase. This includes the addition of CoT information and correction few-shot examples designed for different error types.

Overall, during the preprocessing stage, the inputs are the information from the database and the training set. The outputs are a vector database, Few-shot in the form of Query-CoT-SQL, and the database schema. This process is automated, does not require human intervention, and is entirely driven by an Agent.

3.4 Extraction

The goal of **Extraction** is to prepare the necessary information based on specific NLQs, including schema linking, stored database values, few-shot examples, and any required instructions. This part is decoupled from the specific database query language and is only related to the format of data storage and the auxiliary information needed to solve the problem. For instance, C3-SQL[6] utilizes Clear Prompting (CP) to provide effective prompts, while DIN-SQL[27] employs schema linking and classification & decomposition to categorize and break down the NLQ.

Listing 4: Format of Extraction

```
Input:
/* Database schema */
{db_info}
{rule}
/* Answer the following: {query} */

Output:
{reason}
{column}
{values}
```

The **Extraction** process in OpenSearch-SQL consists of entity extraction, value extraction, and column filtering. After **Extraction**, we use Info Alignment to align the extracted information with the input to produce the final output. We present the format of the extraction prompt and its input and output in the listing 4. We obtain candidate columns and values through LLM and directly extract entities from the NLQ using another simple prompt. The details are outlined below:

Entity Extraction. To find similar values in the database and perform column filtering, we first use the LLM understand and process the NLQ and basic database information to extract potential entities. Then, we organize some predefined entity terms together with these entities to perform the subsequent values retrieval and column filtering.

Values Retrieval. From the extracted entities, we perform vector retrieval to find results with the highest embedding similarity to the entity words. Due to the nature of embedding similarity, this method can prevent mismatches caused by typos or other character-level differences. Additionally, for phrases and longer texts, we perform split retrieval to avoid recall failures due to differences in database storage formats. During the recall phase, we

filter out portions of the top K most similar entities that fall below a specific threshold, retaining them as the final results.

Column Filtering. In detail, we utilize two methods to recall relevant tables and columns. First, we employ a large language model (LLM) to select tables and columns related to the natural language query (NLQ) from complete database information. Then, we use vector retrieval to find columns in the database with similarity to NLQ entities exceeding a certain threshold, integrating these to form a preliminary subset of the final Schema information. Although this multi-path recall method may lack some precision in filtering, it is lighter and more streamlined in process.

Info Alignment. At the end of the extraction phase, we use an Info alignment agent to align the style of the SELECT statement: extracting phrases or clauses from the NLQ that correspond one-to-one with the generated SELECT content. This ensures that both the quantity and order of the SELECT content meet expectations. To avoid misselection caused by columns with the same name in different tables, we expand the schema information by reintegrating the primary keys of each table and all columns with the same name as the columns selected in column filtering into the column filtering results, ensuring the accuracy and comprehensiveness of the selected columns. Finally, all the extracted information and the content of the Alignment will be input into the **Generation**.

3.5 Generation

The definition of **Generation** is: using appropriate methods to drive LLMs to generate specific components and complete SQL. MAC-SQL [36] generates sub-SQLs and then assembles them into the final result, while DAIL-SQL [7] drives LLMs to generate SQL through few-shot learning. Additionally, some methods [18, 23, 34] utilize LLMs during Supervised Fine-Tuning (SFT) or generate multiple sets of SQL to complete tasks.

Listing 5: Example of Generation

```
Input:
{fewshot}
/* Database schema */
{db_info}
{similar values}
{rule}
Rule Example: For parts involving division that
contain integer types, CAST them to REAL

/* Answer the following: How many patients
   with a normal Ig A level came to the
   hospital after 1990? normal Ig A level
   refers to IGA > 80 AND IGA < 500; */
{info alignment}
Info Alignment Example: SELECT content: [How many
patients]

Output:
```

```

#reason: The question wants to count the
  number of patients with a normal Ig A
  level who came to the hospital after
  1990, so SELECT will count distinct
  patients based on the specified
  conditions.
#columns: Patient.ID, Laboratory.IGA,
  Patient.`First Date`
#values: normal Ig A level refers to
  Laboratory.IGA > 80 AND Laboratory.IGA <
  500; came to the hospital after 1990
  refers to strftime('%Y', Patient.`First
  Date`) >= '1990'
#SELECT: How many patients refer to COUNT(
  DISTINCT Patient.ID)
#SQL-like: Show COUNT(DISTINCT Patient.ID)
  WHERE Laboratory.IGA > 80 AND Laboratory
  .IGA < 500 AND YEAR(Patient.`First Date
  `) >= '1990'
#SQL: SELECT COUNT(DISTINCT T1.ID) FROM
  Patient AS T1 INNER JOIN Laboratory AS
  T2 ON T1.ID = T2.ID WHERE T2.IGA > 80
  AND T2.IGA < 500 AND strftime('%Y', T1.`
  First Date`) >= '1990'

```

SQL Generation. As shown in Listing 5, we chose to use the progressive generation + dynamic few-shot to better drive LLMs in generating SQL. Specifically, progressive generation is a customized CoT approach, the format is the same as Listing 2. We first defined SQL-Like: a type of SQL language that ignores specific syntactical elements (such as JOINs and the formatting of functions), aiming to encourage LLMs to focus more on the logic within SQL rather than the formatting. We then instruct the model to sequentially provide the **analysis**, the contents of the **SELECT statement**, **relevant columns** and **values**, as well as the **SQL-Like** and **SQL** outputs. This progressive generation emphasizes the LLMs' understanding of the structure of the generated SQL, while also facilitating the identification of errors in the LLMs' reasoning. For individual NLQs, we allow LLMs to generate multiple candidate SQL queries. For the construction of few-shot examples, the system retrieves the top K_f most similar queries based on MQs, and uses their corresponding Query-CoT-SQL forms as the few-shot examples in the final instruction.

Listing 5 demonstrates the composition of inputs and outputs during the generation phase through an example.

- (1) In **input**, we provide the necessary output requirements and database information by the Rules and Database schema, and we offer similar examples through few-shot in Query-CoT-SQL format. **Similar Values** provide specific values from the database that are relevant to the NLQ, while **Info Alignment** aligns the SELECT style of the dataset with the LLMs.
- (2) In the **output**, we require the following sequential generation: **Reason**: Analysis of NLQ, **Columns**: The relevant

columns in the SQL, **Values**: The related values, **SELECT**: The content of the SELECT statement. Then, generate **SQL-Like** query that ignores syntax formatting and ultimately produces the final SQL queries.

Listing 6: Examples of Alignments

Agent Alignment:

Raw SQL:	SELECT ID FROM table WHERE table. name = 'John'
value in Database:	table.name='JOHN'
Aligned SQL:	SELECT ID FROM table WHERE table. name = 'JOHN'

Function Alignment:

Raw SQL:	SELECT ID FROM table ORDER BY MAX(score)
Aligned SQL:	SELECT ID FROM table GROUP BY ID ORDER BY score

Style Alignment:

Raw SQL:	SELECT ID FROM table ORDER BY score DESC LIMIT 1
Aligned SQL:	SELECT ID FROM table WHERE score IS NOT NULL ORDER BY score DESC LIMIT 1

Alignments. The errors in the SQL query generation process primarily arise from differences in syntax, databases, natural language queries (NLQ), and dataset styles. Additionally, employing large models to sample at higher temperatures for diverse answers may introduce errors and noise. Therefore, we attempt to reduce the biases caused by these discrepancies through an alignment mechanism. Specifically, as shown in the listing 6, the alignment mechanism consists of the following three components:

- Agent Alignment: Ensures that the columns and values from the database are correctly represented in the SQL. If there is a mismatch, corrections are made. A common example is a WHERE condition in SQL that does not match the information stored in the database.
- Function Alignment: Standardizes the SQL aggregate functions to prevent errors caused by incorrect expressions. This includes handling inappropriate AGG functions, nestings, and redundant JOINs.
- Style Alignment: Addresses issues related to dataset characteristics, such as the use of IS NOT NULL and the choice between MAX and LIMIT 1.

After completing the above steps, the SQL is executed, and during the **Refinement**, it is optimized and the final selection is made based on the execution results.

3.6 Refinement

The work of **Refinement** involves optimizing and selecting the generated SQLs. This includes correcting errors based on execution results, and selecting the best answer from multiple candidate SQLs.

Due to the variability of results generated by LLMs and the frequent issues with instruction adherence in complex tasks, we

propose a consistency alignment mechanism based on checking to enhance the quality of SQL. As illustrated in Figure 2, after aligning the generated SQLs, the refinement consists of the following two steps:

- (1) **Correction:** Execute the SQL and fix it based on the error details in the execution results, such as syntax errors or empty results. Each type of error corresponds to different few-shot and error-correction instructions. The purpose of this step is to avoid issues such as lack of results or execution failures caused by minor details.
- (2) **Self-consistency & vote:** Exclude SQLs that cannot be fixed and those that result in empty answers and choose the option with the highest output consistency in the execution results. Additionally, among SQL queries with the same answers, we select the one with the shortest execution time. This approach achieves a dual enhancement in both execution results and execution speed for the final selected SQL. Given the set $K = \{R_1, R_2, \dots, R_n\}$, where $R_i = (SQL_i, ans_i, t_i)$. SQL_i represents the SQL query, ans_i represents the result of the execution, and t_i represents the execution time. We obtain the final SQL query using:

```
SELECT SQL FROM K WHERE ans =
(SELECT ans FROM K
 GROUP BY ans ORDER BY
 COUNT(*) DESC LIMIT 1)
ORDER BY t LIMIT 1
```

(3)

It is worth mentioning that self-consistency often leads to higher costs. In the evaluation, when only one SQL is generated in the **Generation** and the self-consistency mechanism is not used, OpenSearch-SQL, v2, still ranks first in the BIRD benchmark.

3.7 Algorithm

In this section, we provide an overall summary of OpenSearch-SQL and present the entire framework in Algorithm 1. In this algorithm, we take an NLQ as an example, encompassing each specific step from preprocessing to the final generation of SQL.

It can be seen that we first process the database table by table, field by field, during preprocessing to build an index database D_v of values and columns' names, while also generating the original schema S of the database. Next, for the Query-SQL pairs in the training set, we use M_L to generate CoT information to construct the new few-shot set F . In the main function, we first process the NLQ Q using a LLM. Through A_E we obtain the entity information E_Q and select the column $t.c_Q$. Then, we retrieve similar values v_Q and expand the database information $t.c_E$. These details are integrated through Alignment A_a to create a schema S_Q tailored for Q . The next step is to obtain the few-shot f_Q in Query-CoT-SQL format relevant to the question based on $MQs(Q)$. Next, SQLs are generated through A_G and aligned using A_a to obtain $SQLs_A$. Finally, execution errors are corrected with A_R , and the SQL with the highest consistency and shortest execution time, SQL_R , is selected as the final answer.

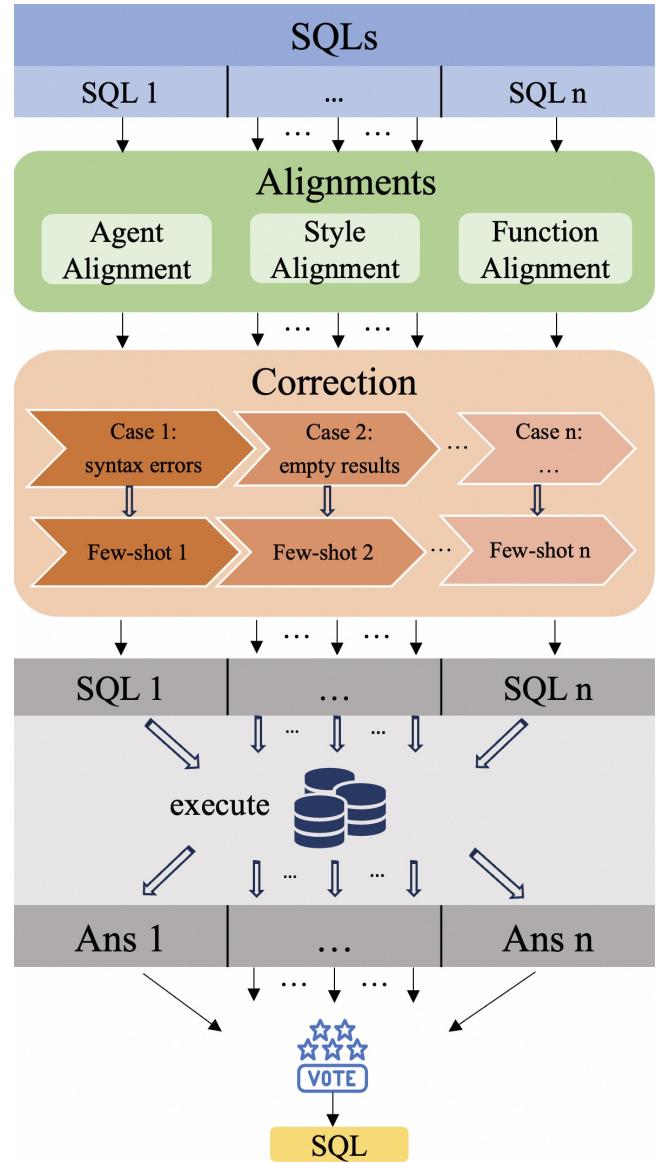


Figure 2: Architecture of the Refinement: Optimizing SQLs through Correction and Consistency & Vote to get the best SQL.

3.8 Optimization

OpenSearch-SQL has achieved significant results, yet there is still plenty of room for optimization. Currently, we haven't fully focused on the detailed adjustments of prompts and the precise selection of columns and values in OpenSearch-SQL. Moreover, in the generation tasks, we only use a single prompt as the instruction for the LLM to generate a SQL candidate set, without additional optimization. On the other hand, the SFT model shows great potential in improving Text-to-SQL performance. Research such as CHESS [34], MCS-SQL [18] and distillery [23] has already demonstrated

Algorithm 1 Algorithm of OpenSearch-SQL

Input: Target database D, user question Q, train set T

- 1: Extraction Agent A_E , Generation Agent A_G , Refinement Agent A_R , Alignment Agent A_a , vector model M_E , LLM M_L , Fewshot Set F, Database Schema S, vector database D_v
- 2: **for** Table t in D **do** ▷ Preprocessing
- 3: **for** column c , values v in t **do** ▷ Build vector database
- 4: get description d
- 5: **if** v is String **then**
- 6: $D_v \leftarrow M_E(v), M_E(t.c) : t.c$ ▷ Add index to D_v
- 7: **end if**
- 8: $S \leftarrow S + d$
- 9: **end for**
- 10: **end for**
- 11: **for** Query-SQL pair (q_t, s_t) in T **do** ▷ Create fewshot
- 12: $CoT_t \leftarrow M_L((q_t, s_t))$
- 13: Add Query-CoT-SQL pair (q_t, CoT_t, s_t) to F
- 14: **end for**
- 15: **end for** ▷ Main Process
- 16: **for** $E_Q, t.c_Q \leftarrow A_E(Q, S)$
- 17: $v_Q \leftarrow D_v(E_Q)$ ▷ Get similar values to Q
- 18: $t.c_E \leftarrow D_v(E_Q)$ ▷ Get table, column related to Q
- 19: $S_Q \leftarrow A_a(Q, S, t.c_E, t.c_Q, v_Q)$ ▷ New Schema
- 20: $f_Q \leftarrow F(MQs(Q))$
- 21: $SQLs \leftarrow A_G(f_Q, S_Q, Q)$
- 22: $SQLs_A \leftarrow A_d(SQLs, S_Q, Q)$ ▷ Align SQLs to Q and S_Q
- 23: $SQLR \leftarrow A_R(SQLs_A, S_Q)$ ▷ Refinement of SQLs
- 24: **return** $SQLR$

the effectiveness of these methods, suggesting further optimization can be pursued in these directions.

In terms of methods within OpenSearch-SQL, we believe that few-shot approaches are not limited to Query-CoT-SQL pairs; there are other options available, offering potential for enhanced performance. Additionally, the research on alignments is still in its early stages, indicating there is room for further development in improving Text-to-SQL task performance.

4 EXPERIMENTS

In this section, we demonstrate the effectiveness of OpenSearch-SQL through experiments and explore the role of its various module.

4.1 Experimental Setup

Datasets. We show the characteristics of these two datasets in Table 1. Considering the specific queries and SQL complexity of the datasets. Bird has relatively fewer types of databases but with more complex database structures and a higher average difficulty of SQL. In contrast, Spider has a rich variety of databases, but the average difficulty of SQL is relatively lower.

BIRD. [20] (Big Bench for Large-scale Database Grounded Text-to-SQL Evaluation) represents a pioneering, cross-domain dataset that examines the impact of extensive database contents on text-to-SQL parsing. BIRD contains over 12,751 unique question-SQL pairs, 95 big databases with a total size of 33.4 GB. It covers more than 37

Dataset	train	dev	test	domains	databases
Spider	8659	1034	2147	138	200
Bird	9428	1534	1789	37	95

Table 1: Statistics of the datasets.

professional domains, such as blockchain, hockey, healthcare, and education, etc. The BIRD dataset released a cleaner version of the dev set on 4 July 2004. However, for the sake of fairness, we will still evaluate our results using the dev data from before July.

Spider. [44] is a large-scale complex and cross-domain semantic parsing and text-to-SQL dataset annotated by 11 Yale students. The goal of the Spider challenge is to develop natural language interfaces to cross-domain databases. It consists of 10,181 questions and 5,693 unique complex SQL queries on 200 databases with multiple tables covering 138 different domains. In Spider, we perform tuning on the dev set, assuming the test set is not accessible.

Evaluation. Following the evaluation criteria from BIRD [20] and the Spider test suite [47], we assessed two metrics: **Execution Accuracy (EX)** and **Reward-based Valid Efficiency Score (R-VES)**. EX is defined as the proportion of identical execution results between the predicted SQL and the gold SQL. The purpose of R-VES is to measure the execution efficiency of the predicted SQL when performing the same task.

Baselines. We selected the LLM-driven methods from the BIRD and Spider rankings as baselines:

- (1) **GPT-4** [26] employs a zero-shot text-to-SQL prompt for SQL generation.
- (2) **DIN-SQL** [27] divides questions into different types through multiple modules and instructs LLM to generate the final SQL through various prompts.
- (3) **DAIL-SQL** [7] assists LLMs in generating SQL by selecting similar question-SQL pairs for different questions as few-shot examples.
- (4) **MAC-SQL** [36] simplifies the challenges LLMs face by using sub-databases and sub-questions to generate SQL.
- (5) **MCS-SQL** [18] generates multiple sets of SQL using multiple prompts and employs a unified Multiple-Choice Selection (MCS) to select the final SQL.
- (6) **C3-SQL** [6] has constructed a systematic Zero-shot Text-to-SQL approach through three modules: Clear Prompting (CP), Calibration with Hints (CH), and Consistent Output (CO).
- (7) **CHESS** [34] improves the effectiveness of SQL generation by constructing efficient retrieval and schema pruning methods to reduce the interference of redundant information.
- (8) **Distillery** [23] suggests that the importance of schema linking has diminished with the development of LLMs, and the fine-tuned SFT GPT-4o has achieved optimal results.

Implementation Details. To demonstrate the capabilities of OpenSearch-SQL v2, the specific versions of the models we have chosen are GPT-4o-0513 [26] for generation, and bge-large-en-v1.5 [41] for retrieval.

For efficiency and comparability, we conducted ablation experiments on the MIN DEV officially released by BIRD. The purpose of these ablation experiments is to evaluate the generalization ability of OpenSearch-SQL across various models, as well as to verify the effectiveness of each submodule in the method. During the **Extraction**, we set the temperature from 0, in **Generation** and **Refine** stage, the temperature is set to 0.7. The number of few-shot examples is selected from $N = \{0, 3, 5, 7, 9\}$, with a filtering index threshold value of 0.65, and the maximum number of SQL votes is chosen from $K = \{1, 7, 15, 21\}$.

4.2 Main Result

To demonstrate the effectiveness of OpenSearch-SQL, our performance on the Spider and BIRD datasets has achieved state-of-the-art (SOTA) levels, highlighting the stability and superiority of our approach. In Table 3, we compare the effectiveness of our method with the baseline methods in the BIRD dataset. Furthermore, Table 2 presents the experimental results on the Spider dataset. Since the Spider leaderboard will no longer accept new submissions starting February 2024, the results are composed of current leaderboard data and those reported in relevant papers. Notably, the BIRD dataset underwent a cleansing of the development set in July, which might lead to discrepancies in comparisons. To ensure comparability with previous evaluations, we have continued to use the dataset before July for our experiments.

BIRD Results. We present the performance of Baseline and OpenSearch-SQL, v2 on the BIRD dataset in Table 2. The results in the table are derived entirely from the BIRD leaderboard. At the time of submission, our method achieved 69.3% EX on the BIRD development set and 72.28% EX and 69.36% R-VES on the BIRD holdout test set, all ranking first. It is noteworthy that our method did not involve any fine-tuning to customize the LLM. Meanwhile, we demonstrated the performance of OpenSearch-SQL without applying Self-Consistency & Vote. It can be seen that even using a single generation SQL, OpenSearch-SQL achieved an EX of 67.8, still maintains the top performance on the development set.

In Figure 3, we compare the performance of our method across various difficulty levels. It is evident that Self-Consistency & Vote shows the most significant improvement for difficult problems, achieving an absolute difference of 7.64%. However, there is no notable difference for easy and medium problems. This indicates that as problem difficulty increases, the susceptibility of large models to hallucinations also rises.

Spider Results. To demonstrate the generalizability of OpenSearch-SQL, we also evaluated its performance on the Spider test set. We used the default configuration of our method, making no other changes, except for necessary adjustments to accommodate the data set format. Since the Spider leaderboard is no longer updated and the test set is publicly available, we extracted results from the baseline method’s paper and leaderboard, with results extracted from the paper marked with a *. Table 3 shows the performance of

Method	EX		R-VES
	dev	test	test
GPT-4	46.35	54.89	51.57
DIN-SQL + GPT-4	50.72	55.90	53.07
DAIL-SQL + GPT-4	54.76	57.41	54.02
MAC-SQL + GPT-4	57.56	59.59	57.60
MCS-SQL + GPT-4	63.36	65.45	61.23
CHESS	65.00	66.69	62.77
Distillery + GPT-4o(ft)	67.21	71.83	67.41
OpenSearch-SQL + GPT-4	66.62	-	-
OpenSearch-SQL + GPT-4o w/o Self-Consistency & Vote	67.80	-	-
OpenSearch-SQL + GPT-4o	69.30	72.28	69.36

Table 2: Execution accuracies (EX) and Reward-based Valid Efficiency Scores (R-VES) on the BIRD dev and test sets. The result is taken from the leaderboard.

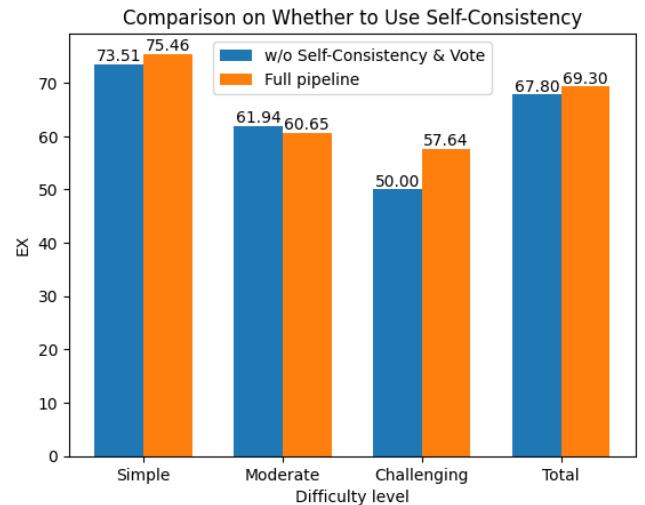


Figure 3: Comparison of the impact of using Consistency & Vote across different difficulty levels.

the methods, where our method achieved an execution accuracy of 87.1%.

4.3 Modular Ablation

Table 4 provides a detailed overview of the Execution Accuracy (EX) when each module is removed from the method. We specifically examine the roles of the **Extraction**, **Generation**, **Refinement**,

Method	EX
GPT-4	83.9
C3+ChatGPT	82.3
DIN-SQL	85.3
DAIL-SQL+GPT-4	86.6
MAC-SQL+GPT-4	82.8*
MCS-SQL+GPT-4	89.6*
CHESS	87.2*
Distillery + GPT-4o(ft)	-
OpenSearch-SQL+GPT-4	86.8
OpenSearch-SQL+GPT-4o	87.1

Table 3: Execution accuracy (EX) for the Spider test sets. * denotes that the result is taken from their paper rather than from the leaderboard.

and **Alignment** modules. Furthermore, we investigate the impact of key submodules such as Retrieve, Few-shot, Chain of Thought (CoT), and Self-consistency & Vote. Notably, Self-consistency & Vote functions as the final step in the selection among multiple SQLs. To more clearly illustrate the effect of each module on the optimization of individual SQL, we present the EX results of this portion separately.

To thoroughly demonstrate the roles of each module in the method, we first recorded the Execution Accuracy (EX) of the SQLs ultimately generated by OpenSearch-SQL. Given that the Consistency & Vote mechanism selects from multiple SQLs, it may influence the evaluation of the impact of each module on an individual SQL. Therefore, we also recorded the accuracy of the execution of the SQLs obtained at the generation stage (EX_G) and prior to the Consistency & Vote stage (EX_R). These two metrics directly capture the effect of each module on individual SQLs, providing a more precise assessment of each module’s influence.

As shown in the table 4, from left to right, we present EX_G , EX_R , EX. From top to bottom, the roles of the Extraction, Generation, Refinement, and Alignment modules are recorded in sequence. From the table, we can observe:

- The EX of SQLs increases monotonically as the workflow progresses, indicating that different modules have distinct positive impacts on SQL optimization, confirming the necessity of each module.
- Without Extraction, the complete database schema is used directly, bypassing values retrieval and column filtering. It can be observed that the combined improvement from values retrieval and column filtering for a single SQL is approximately equal to the improvement provided by Extraction, suggesting minimal overlap in their roles.

- Few-shot demonstrates a significant effect during the generation phase and further enhances the final SQL performance during the Refinement phase, highlighting the importance of a dynamic Few-shot approach. We speculate that few-shot can greatly enhance the model’s potential, while also improving the stability of the generated results. Further analysis of few-shot will be conducted in the experiments 4.4.
- During the **Generation**, CoT and Few-shot significantly enhance individual SQL results, indicating a positive correlation between the amount of information in the prompt and the effectiveness of LLM-generated SQL. However, after the **Refinement** and Self-Consistency & Vote stages, the gains from CoT are notably reduced. We believe this suggests that CoT does not significantly enhance the model’s foundational generation capabilities but instead improves the stability of the results. Additionally, it is observed that **Refinement** provides +3.2% to results generated without CoT, highlighting its crucial role in increasing result stability.
- The Alignment module has played a positive role at various stages, significantly enhancing the performance of individual SQL, thereby demonstrating its unique optimization potential. Specifically, Alignments applied after the generation phase can assist the Correction module in accurately fixing erroneous SQL, rather than merely making them executable.
- It is evident that Self-Consistency & Vote consistently raise the performance ceiling of LLMs. Based on our observations of the generated SQL, this mechanism primarily helps avoid errors caused by low-probability randomness.

4.4 Few-shot

In Table 5, we explore the impact of different dynamic Few-shot strategies on Text-to-SQL. First, during the generation phase, we compare the Few-shot effects of the classic Query-SQL pair with our Query-COT-SQL pair, and we evaluate the scenario without Few-shot. We also investigate the influence of Few-shot in other stages. From the table, we can summarize the following points:

- The Few-shot strategy significantly raises the upper limit of SQL accuracy generated by LLMs. Whether in the form of a Query-SQL Pair or a Query-CoT-SQL Pair, compared to without few-shot, these formats significantly enhance the performance of LLMs at each stage. However, the improvement from few-shot in the **Refinement** stage is relatively smaller.
- During the **Generation**, for a single SQL query, the Few-shot strategy provides the greatest performance improvement, with the Query-CoT-SQL pair achieving a 3.0% absolute EX increase compared to the Query-SQL pair form, and a 6.4% increase in EX compared to the scenario without Few-shot.
- In **Refine** and under the influence of **self-consistency & vote**, the gap between different forms of Few-shot gradually narrows, indicating that there is some redundancy in the SQL optimization content in different stages.
- Although the Few-shot strategy has limited optimization effects on individual SQL queries during the **Refine**, it can

Pipeline Setup	EX_G	ΔEX_G	EX_R	ΔEX_R	$EX_R - EX_G$	EX	ΔEX	$EX - EX_R$
Full pipeline	65.8	-	68.2	-	+2.4	70.6	-	+2.4
w/o Extraction	61.6	-4.2	66.2	-2.0	+4.6	67.4	-3.2	+1.4
w/o Values Retrieval	64.4	-1.4	66.6	-1.6	+2.2	69.2	-1.4	+2.6
w/o column filtering	63.2	-2.6	65.0	-3.2	+1.8	68.6	-2.0	+3.6
w/o Info Alignment	62.8	-3.0	67.6	-0.6	+4.8	68.6	-2.0	+1.0
w/o Generation	-	-	-	-	-	-	-	-
w/o Few-shot	60.4	-5.4	63.0	-5.3	+2.6	66.0	-4.6	+3.0
w/o CoT	63.0	-2.8	66.2	-2.0	+3.2	69.2	-1.4	+2.6
w/o Alignments	65.8	-	67.0	-1.2	+1.2	69.6	-1.0	+2.6
w/o Refinement	65.8	-	67.0	-1.2	+1.2	67.0	-	-
w/o Correction	65.8	-	67.0	-1.2	+1.0	69.8	-0.8	+2.8
w/o Few-Shot	65.8	-	67.6	-0.6	+1.6	69.4	-1.2	+1.8
w/o Self-Consistency & Vote	65.8	-	68.2	-	-	68.2	-	-

Table 4: The execution accuracy (EX) of the pipeline by removing each component on the Mini-Dev set. EX_G represents the accuracy of a single SQL obtained from the Generation, EX_R indicates the performance of a single SQL before Self-Consistency & voting, and EX refers to the results after all processes are completed.

Method	EX_G	ΔEX_G	EX_R	ΔEX_R	EX	ΔEX
Query-CoT-SQL pair Few-shot	65.8	-	68.2	-	70.6	-
w/o Few-shot of Generation	59.6	-6.2	63.0	-5.2	66.0	-4.6
w Query-SQL pair Few-shot of Generation	63.0	-2.8	66.2	-2.0	69.2	-1.4
w/o Few-shot of Refinement	65.8	-	67.6	-0.6	69.4	-1.2
w/o Few-shot of Generation & Refinement	59.6	-6.2	62.8	-5.4	66.0	-4.6

Table 5: Few-shot Performance Comparison Results.

improve the final EX of consistency and voting, suggesting that the Few-shot-driven SQL correction results exhibit higher consistency with one another.

4.5 Self-Consistency & Vote

In our result, we used a single prompt to generate 21 results as answers. To investigate the optimal number of SQL outputs generated by beam search for self-consistency with a single prompt, we designed the experiment shown in the figure 4.

We conducted experiment with GPT-4o and GPT-4o-mini to compare the performance of LLMs with different parameter levels, and the number of candidates is taken from $N = \{1, 3, 7, 15, 21\}$. As observed in the figure, the EX value for GPT-4o consistently increases with the number of candidates. In contrast, GPT-4o-mini achieves optimal results with 7 and 15 candidates. This indicates that for models with sufficiently large parameters, the number of generated answers can be safely increased to achieve self-consistency & vote. However, for smaller models, the number of outputs should be controlled to prevent negative impacts on self-consistency due to an excessive number of answers.

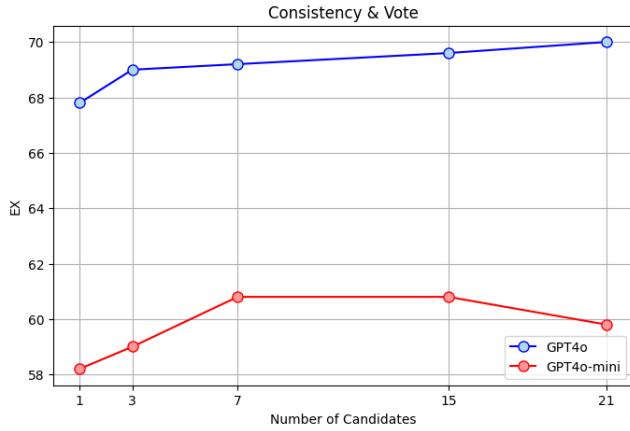


Figure 4: Comparison of the impact of different numbers of candidates on model performance

4.6 Execution Cost

In Table 6, we provide a simple estimate of the time and resource consumption for using OpenSearch-SQL with GPT-4o. Generally, the execution time for the entire process primarily depends on large language model (LLM) calls, retrieval time, and SQL execution time. In practice, using HNSW [24] can significantly reduce retrieval latency, thus shifting the main time consumption to LLM responses and SQL execution.

Specifically, the tokens for extraction and generation are mainly derived from the database schema and a few examples. The generation phase takes longer since it uses BeamSearch, which generates more tokens than the extraction phase. Apart from the SELECT alignment, which is triggered every time, other alignment steps are only activated when there are issues with the SQL, so they are not time-consuming in most cases. As for the refinement phase, the time is mainly due to SQL execution and LLM corrections, while Self-consistency & Vote contribute minimally to the time consumption.

4.7 CoT

In this section, we provide a brief analysis of the CoT method we designed and compare the performance of structured CoT designed for OpenSearch-SQL with the unstructured CoT approach (i.e., "let's think step by step") as well as scenarios without CoT. To

Modular	Time(s)	Token
Extraction	4-9	5000-10000
	Entity & Column	5000-10000
	Retrieval	-
Generation	5-15	4000-8000
Refinement	0-25	0-5000
	Correction	0-5000
	Self-consistency & Vote	<0.01s
Alignments	0-15	500-2000
SELECT Alignment	1-3	500-600
Agent Alignment	0-7	100-500
Style Alignment	0-5	100-500
Function Alignment	0-4	100-500
Pipeline	7-60s	9000-25000

Table 6: Exection Cost Performance

isolate the impact of CoT, we eliminate few-shot learning during the **Generation** phase and focus solely on the execution accuracy when generating a single SQL(EX_G) and multiple SQLs with Self-consistency & Vote(EX_V). This allows for a straightforward assessment of CoT's direct impact on SQL generation.

As shown in Table 7, it is apparent that utilizing CoT leads to a significant improvement in SQL generation compared to not using CoT. Moreover, structured CoT demonstrates more substantial gains than unstructured CoT. Furthermore, examining the results of $EX_V - EX_G$, we observe that the implementation of **Self-consistency & Vote** provides even greater improvements than simply using CoT alone, with structured CoT achieving a higher relative enhancement.

In summary, structured CoT achieved the optimal results. When sampling multiple answers for voting, structured CoT exhibits a more significant relative improvement compared to other methods.

Modular	EX_G	EX_V	$EX_V - EX_G$
w/o CoT	57.6	59.2	1.6
Unstructured CoT	58.2	63.0	4.8
Structured CoT	58.8	65.0	6.2

Table 7: CoT Performance Comparison Results

5 RELATED WORK

For a long time, computer professionals have been hoping to convert natural language queries into specific database queries according to certain scenarios, in order to dramatically reduce the learning and usage costs for individuals, bridge the gap between laypersons and expert users, and enhance the efficiency of using, analyzing, and querying databases. The Text-to-SQL task is one of the most representative examples of these, and success in the Text-to-SQL task

would imply that similar methods could assist in various SQL-like tasks. However, this task continues to be challenging, particularly due to the strong correlation between the SQL structure and the database structure, as well as the limitations imposed by the expression format requirements[17].

5.1 Classic Method

Classical methods often do not directly generate SQL but assemble it through various means. These methods handle components like SELECT, WHERE, etc., separately to construct SQL clauses. An example of such a method is SQLNET[42], which employs a sketch-based approach where the sketch contains a dependency graph, allowing one prediction to take into account only the previous predictions it depends on. RYANSQL[4] and SyntaxSQLNet[43] have tried to generalize sketch-based decoding, attempting not only to fill in the gaps of a sketch but also to generate the appropriate sketch for a given text. Additionally, there are methods that utilize graph representation[2, 3] and employ intermediate languages[12, 31] to enhance the capabilities of LLMs. These methods may be constrained by the inherent capabilities of the models, which may lead to inferior performance. For the Text-to-SQL task, their theoretical value outweighs their practical value.

5.2 Large Language Models

Large Language Models (LLMs) represent a significant breakthrough in the field of artificial intelligence in recent years. They are based on deep learning technology, typically built on the Transformer architecture. These models consist of large-scale neural networks capable of training on massive amounts of text data, thereby endowing them with the powerful ability to generate and understand natural language text.

Throughout the development of LLMs, several notable models have emerged, the most famous being GPT (Generative Pre-trained Transformer) [30] and BERT (Bidirectional Encoder Representations from Transformers) [5]. GPT excels in natural language generation, while BERT is exceptional in understanding natural language. These models are widely applied across various fields, such as text generation, information extraction, and machine translation, effectively advancing natural language processing.

Recently, more advanced models have emerged, such as GPT-4 [26], LLAMA [25], Qwenmax [1], and DeepSeekV2.5 [21], which features a mixture of experts (MoE) architecture . These emerging models can be regarded as significant milestones in the fields of natural language processing and machine learning due to their outstanding performance in a variety of downstream tasks. Thanks to the excellent capabilities of these models, natural language processing now plays an increasingly important role in addressing many complex and meaningful issues. As technology continues to advance, large language models are expected to have a profound impact on scientific research, commercial applications, and daily life worldwide.

5.3 LLM-based Agents and RAG

With the advent of the era of large models, the enhanced capabilities of large models have simplified many previously difficult tasks. The LLM-based agent is an approach that leverages the power of

large models to carry out a multitude of complex tasks[39], usually performing tasks by assuming specific roles, such as programmers, educators, and domain experts[29]. Retrieval-Augmented Generation (RAG) is a type of agent that assists in answering questions by retrieving highly relevant documents[9]. Furthermore, multiple RAG techniques, including Modular RAG[32, 45] and Advanced RAG[8, 22], assist LLMs in providing more accurate and richly formatted answers by employing techniques such as re-ranking and rewriting.

5.4 LLM for Text-to-SQL

Recent research shows that the most effective Text-to-SQL methods primarily rely on multi-Agent algorithms based on LLMs. These methods aim to provide the most helpful information to the LLM through Agents to assist in predicting SQL. In this process, the classic Text-to-SQL task benchmarks have gradually evolved and become more complex, with commonly used Text-to-SQL evaluation tasks progressively shifting from WiKiSQL[48] to Spider[44] and BIRD[20]. In these tasks, researchers[33] enhance the LLM's Text-to-SQL abilities through chain-of-thought reasoning. C3[6] has developed a systematic zero-shot Text-to-SQL method that includes several key components. DIN-SQL[27] addresses the Text-to-SQL task by distinguishing question difficulty and adjusting specific prompts accordingly. DAIL-SQL[7] assists the model in generating SQL by retrieving structurally similar questions or SQL as few-shot examples, while MAC-SQL improves capabilities through task decomposition and recombination. Meanwhile, some studies have opted for a fine-tuning + Agent approach to accomplish the Text-to-SQL task, such as Dubo-SQL[35], SFT CodeS[19], DTS-SQL[28]. Notably powerful methods currently include MCS-SQL[18], which enhances the stability of Text-to-SQL tasks by using multiple sets of prompts for voting. CHESS[34], which simplifies the generation difficulty through an excellent column filtering mechanism, and Distillery[23], which optimizes the task by fine-tuning GPT-4O.

Overall, as the capabilities of large models continue to improve, the framework for the Text-to-SQL task has gradually converged and standardized into the form of LLMs + Agents. Meanwhile, the capabilities of Text-to-SQL methods are also constantly enhancing, becoming tools that can truly help simplify people's work in everyday applications.

6 CONCLUSION

In this paper, we introduce OpenSearch-SQL, a method that enhances the performance of the Text-to-SQL task using dynamic Few-shot and consistency alignment mechanisms. To improve the model's handling of prompts, we extend the original examples with a LLM and supplement them with CoT information, forming a Query-CoT-SQL Few-shot configuration. To the best of our knowledge, this is the first exploration of using LLMs to extend Few-shot CoT content in the Text-to-SQL task. Additionally, we developed a novel method based on a consistency alignment mechanism to mitigate hallucinations, enhancing the quality of agent outputs by reintegrating their inputs and outputs. Our approach does not rely on any SFT tasks and is entirely based on directly available LLMs and retrieval models, representing a pure architecture upgrade for Text-to-SQL. As a result, at the time of submission, we achieved the

top position in all three metrics on the BIRD leaderboard, which demonstrates the significant advantages of our approach.

Our source code will be made available soon, we hope that this Few-shot construction method and consistency alignment-based workflow will offer new perspectives and positive impacts for Text-to-SQL and other multi-agent collaborative tasks.

REFERENCES

- [1] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).
- [2] Ben Bogin, Matt Gardner, and Jonathan Berant. 2019. Global reasoning over database structures for text-to-sql parsing. *arXiv preprint arXiv:1908.11214* (2019).
- [3] Ursin Brunner and Kurt Stockinger. 2021. Valuenet: A natural language-to-sql system that learns from database information. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2177–2182.
- [4] DongHyun Choi, Myeong Cheol Shin, EungGyun Kim, and Dong Ryeol Shin. 2021. Ryansql: Recursively applying sketch-based slot fillings for complex text-to-sql in cross-domain databases. *Computational Linguistics* 47, 2 (2021), 309–332.
- [5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805* [cs.CL] <https://arxiv.org/abs/1810.04805>
- [6] Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Jinshu Lin, Dongfang Lou, et al. 2023. C3: Zero-shot text-to-sql with chatgpt. *arXiv preprint arXiv:2307.07306* (2023).
- [7] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. Text-to-sql empowered by large language models: A benchmark evaluation. *arXiv preprint arXiv:2308.15363* (2023).
- [8] Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. 2022. Precise zero-shot dense retrieval without relevance labels. *arXiv preprint arXiv:2212.10496* (2022).
- [9] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. 2024. Retrieval-Augmented Generation for Large Language Models: A Survey. *arXiv:2312.10997* [cs.CL]
- [10] Chunxi Guo, Zhiliang Tian, Jintao Tang, Pancheng Wang, Zhihua Wen, Kang Yang, and Ting Wang. 2023. Prompting GPT-3.5 for Text-to-SQL with Desemanticization and Skeleton Retrieval. *arXiv:2304.13301* [cs.CL] <https://arxiv.org/abs/2304.13301>
- [11] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards complex text-to-sql in cross-domain database with intermediate representation. *arXiv preprint arXiv:1905.08205* (2019).
- [12] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards complex text-to-sql in cross-domain database with intermediate representation. *arXiv preprint arXiv:1905.08205* (2019).
- [13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *arXiv:1512.03835* [cs.CV] <https://arxiv.org/abs/1512.03835>
- [14] Vagelis Hristidis and Yannis Papakonstantinou. 2002. Discover: Keyword search in relational databases. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 670–681.
- [15] Vagelis Hristidis, Yannis Papakonstantinou, and Luis Gravano. 2003. Efficient IR-style keyword search over relational databases. In *Proceedings 2003 VLDB Conference*. Elsevier, 850–861.
- [16] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. *arXiv preprint arXiv:1704.08760* (2017).
- [17] George Katsogiannis-Meimarakis and Georgia Koutrika. 2023. A survey on deep learning approaches for text-to-SQL. *The VLDB Journal* 32, 4 (2023), 905–936.
- [18] Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2024. MCS-SQL: Leveraging Multiple Prompts and Multiple-Choice Selection For Text-to-SQL Generation. *arXiv preprint arXiv:2405.07467* (2024).
- [19] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024. Codes: Towards Building Open-source Language Models for Text-to-SQL. *arXiv:2402.16347* [cs.CL]
- [20] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2024. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems* 36 (2024).
- [21] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Daya Guo, et al. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434* (2024).
- [22] Xinbei Ma, Yeyun Gong, Pengcheng He, Hai Zhao, and Nan Duan. 2023. Query rewriting for retrieval-augmented large language models. *arXiv preprint arXiv:2305.14283* (2023).
- [23] Karime Maamari, Fadhil Abubaker, Daniel Jaroslawicz, and Amine Mhedhbi. 2024. The Death of Schema Linking? Text-to-SQL in the Age of Well-Reasoned Language Models. *arXiv:2408.07702* [cs.CL] <https://arxiv.org/abs/2408.07702>
- [24] Yu. A. Malkov and D. A. Yashunin. 2018. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *arXiv:1603.09320* [cs.DS] <https://arxiv.org/abs/1603.09320>
- [25] Meta. 2024. Introducing Meta Llama 3: The most capable openly available LLM to date. <https://ai.meta.com/blog/meta-llama-3/>. Accessed: 2023-05-20.
- [26] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, et al. 2024. GPT-4 Technical Report. *arXiv:2303.08774* [cs.CL] <https://arxiv.org/abs/2303.08774>
- [27] Mohammadmreza Pourreza and Davood Rafiei. 2024. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Advances in Neural Information Processing Systems* 36 (2024).
- [28] Mohammadmreza Pourreza and Davood Rafiei. 2024. DTS-SQL: Decomposed Text-to-SQL with Small Large Language Models. *arXiv:2402.01117* [cs.CL]
- [29] Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. 2023. Communicative agents for software development. *arXiv preprint arXiv:2307.07924* (2023).
- [30] Alex Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.
- [31] Ohad Rubin and Jonathan Berant. 2020. SmBoP: Semi-autoregressive bottom-up semantic parsing. *arXiv preprint arXiv:2010.12412* (2020).
- [32] Zhihong Shao, Yeyun Gong, Yelong Shen, Minlie Huang, Nan Duan, and Weizhu Chen. 2023. Enhancing retrieval-augmented large language models with iterative retrieval-generation synergy. *arXiv preprint arXiv:2305.15294* (2023).
- [33] Chang-You Tai, Ziru Chen, Tianshu Zhang, Xiang Deng, and Huan Sun. 2023. Exploring Chain-of-Thought Style Prompting for Text-to-SQL. *arXiv:2305.14215* [cs.CL]
- [34] Shayan Talaei, Mohammadmreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. CHESS: Contextual Harnessing for Efficient SQL Synthesis. *arXiv:2405.16755* [cs.LG] <https://arxiv.org/abs/2405.16755>
- [35] Dayton G Thorpe, Andrew J Duberstein, and Ian A Kinsey. 2024. Dubo-SQL: Diverse Retrieval-Augmented Generation and Fine Tuning for Text-to-SQL. *arXiv preprint arXiv:2404.12560* (2024).
- [36] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Qian-Wen Zhang, Zhao Yan, and Zhoujun Li. 2023. Mac-sql: Multi-agent collaboration for text-to-sql. *arXiv preprint arXiv:2312.11242* (2023).
- [37] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2019. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. *arXiv preprint arXiv:1911.04942* (2019).
- [38] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 452–466.
- [39] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science* 18, 6 (2024), 1–26.
- [40] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *arXiv:2201.11903* [cs.CL] <https://arxiv.org/abs/2201.11903>
- [41] Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. 2023. C-Pack: Packaged Resources To Advance General Chinese Embedding. *arXiv:2309.07597* [cs.CL]
- [42] Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436* (2017).
- [43] Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. 2018. Syntaxqlnet: Syntax tree networks for complex and cross-domain text-to-sql task. *arXiv preprint arXiv:1810.05237* (2018).
- [44] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887* (2018).
- [45] Wenhao Yu, Dan Iter, Shuohang Wang, Yichong Xu, Mingxuan Ju, Soumya Sanyal, Chengguang Zhu, Michael Zeng, and Meng Jiang. 2022. Generate rather than retrieve: Large language models are strong context generators. *arXiv preprint arXiv:2209.10063* (2022).
- [46] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. 2023. Siren's song in the AI ocean: a survey on hallucination in large language models. *arXiv preprint arXiv:2309.01219* (2023).
- [47] Ruiqi Zhong, Tao Yu, and Dan Klein. 2020. Semantic Evaluation for Text-to-SQL with Distilled Test Suites. *arXiv:2010.02840* [cs.CL]

- [48] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103* (2017).
- Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009