

AutoLink: Autonomous Schema Exploration and Expansion for Scalable Schema Linking in Text-to-SQL at Scale

Ziyang Wang^{1*}, Yuanlei Zheng^{1*}, Zhenbiao Cao¹, Xiaojin Zhang², Zhongyu Wei³, Pei Fu⁴, Zhenbo Luo⁴, Wei Chen^{1†}, Xiang Bai¹

¹School of Software Engineering, Huazhong University of Science and Technology

²School of Computer Science and Technology, Huazhong University of Science and Technology

³School of Data Science, Fudan University

⁴MiLM Plus, Xiaomi Inc.

wangziyang@hust.edu.cn, lemuria_chen@hust.edu.cn

Abstract

For industrial-scale text-to-SQL, supplying the entire database schema to Large Language Models (LLMs) is impractical due to context window limits and irrelevant noise. Schema linking, which filters the schema to a relevant subset, is therefore critical. However, existing methods incur prohibitive costs, struggle to trade off recall and noise, and scale poorly to large databases. We present **AutoLink**, an autonomous agent framework that reformulates schema linking as an iterative, agent-driven process. Guided by an LLM, AutoLink dynamically explores and expands the linked schema subset, progressively identifying necessary schema components without inputting the full database schema. Our experiments demonstrate AutoLink’s superior performance, achieving state-of-the-art strict schema linking recall of **97.4%** on Bird-Dev and **91.2%** on Spider-2.0-Lite, with competitive execution accuracy, i.e., **68.7%** EX on Bird-Dev (better than CHESS) and **34.9%** EX on Spider-2.0-Lite (ranking 2nd on the official leaderboard). Crucially, AutoLink exhibits **exceptional scalability, maintaining high recall, efficient token consumption, and robust execution accuracy** on large schemas (e.g., over 3,000 columns) where existing methods severely degrade—making it a highly scalable, high-recall schema-linking solution for industrial text-to-SQL systems.

Code — <https://github.com/wzy416/AutoLink>

Introduction

Text-to-SQL translates natural-language questions into executable SQL over a given database schema, lowering the barrier for non-experts (Katsogiannis-Meimarakis and Koutrika 2023; Li et al. 2024a). Recent systems rely on autoregressive LLMs: the question and a structured schema representation (e.g., *table/column names, descriptions, and primary/foreign keys*) are fed into the model, which then generates the SQL sequence (Shi et al. 2025; Hong et al. 2025). However, in large industrial databases, supplying the entire schema S_{full} introduces **substantial noise from irrelevant elements and the risk of exceeding context window limits**, hindering correct SQL generation (Lei et al. 2025).

*These authors contributed equally.

†Corresponding author.

To address these limitations, **Schema Linking** emerges as a critical sub-task. Schema linking aims to identify a relevant subset of schema elements ($S_{\text{linked}} \subset S_{\text{full}}$) that are necessary to answer the user’s question, thereby reducing the input context and mitigating noise for the subsequent SQL generation module (Wang et al. 2020). The effectiveness of schema linking is often measured by its **strict recall rate (SRR)** (Cao et al. 2024), defined as the proportion of ground-truth schema elements that are successfully included in S_{linked} . A high SRR is paramount, as missing essential schema elements directly limits the upper bound of SQL generation accuracy.

Existing schema linking methods include discriminative scoring of individual tables/columns, e.g., cross-encoders (Li et al. 2023a) or LLM-based scoring (Talaei et al. 2024), whole-schema reasoning and selection, e.g., full-schema prompting and backward/two-stage pipelines (Lee et al. 2025; Yang et al. 2024), graph-based modeling of question–schema structure (Li et al. 2023b), and dual-encoder retrieval as a front-end to accelerate candidate generation; however, these routes share scalability drawbacks in industrial-scale settings because **computation** and **context windows** become bottlenecks, and achieving high SRR typically requires large candidate sets that reintroduce noise and inflate token usage, undermining the goal of schema linking.

To overcome these challenges, we introduce **AutoLink**, a novel schema linking method that redefines the problem as an interactive, sequential discovery process. Inspired by human database engineers’ exploratory workflow, AutoLink employs a large language model powered autonomous agent to dynamically identify and progressively build the relevant schema subset for a natural language question. Crucially, the agent operates without requiring input of the entire database schema. It achieves this by interacting with two specialized environments: one for *direct database exploration* and another for *efficient semantic schema search*. Through a multi-turn dialogue, the agent strategically utilizes a diverse set of actions, including *schema exploration, semantic retrieval, schema verification, and schema expansion*—to iteratively refine the linked schema. This iterative and exploratory ap-

proach enables AutoLink to accurately pinpoint necessary schema elements while effectively filtering out irrelevant information, providing a highly recall schema for subsequent SQL generation.

We rigorously evaluate **AutoLink** on challenging, large-scale text-to-SQL benchmarks, Spider 2.0-Lite and Bird dataset. Our experiments demonstrate that AutoLink significantly advances schema linking, achieving state-of-the-art SRR while maintaining superior token efficiency. Notably, AutoLink achieves an SRR of **91.2%** on Spider 2.0-Lite, dramatically outperforming baselines and maintaining high recall. This robust scalability is coupled with the lowest average token usage across all database scales, stemming from its iterative schema expansion. Furthermore, our findings reveal a critical correlation: higher SRR directly translates to improved SQL execution accuracy (EX) in downstream tasks. AutoLink consistently achieves competitive EX (e.g., **34.92%** on Spider 2.0-Lite and **68.71%** on Bird), often with superior token efficiency, such as requiring less than half the tokens compared to leading approaches on Spider 2.0-Lite. Ablation studies further confirm the critical contribution of each interactive action—particularly semantic retrieval—to AutoLink’s robust performance and its ability to effectively handle complex, real-world industrial databases.

Related Work

Prior work on schema linking falls into two broad families. **Element-level schema linking** scores individual tables or columns for a given question, typically via cross-encoders or LLM-based rankers; representative methods include RESDSQL (Li et al. 2023a), CodeS (Li et al. 2024b), and CHESS (Talaei et al. 2024). **Database-level schema linking** reasons over the entire schema and the question, with three common lines: **(i) full-schema prompting and multi-prompt aggregation**—DIN-SQL (Pourreza and Rafiei 2023), MCS-SQL (Lee et al. 2025), C3 (Dong et al. 2023), DAIL-SQL (Gao et al. 2024), E-SQL (Caferoglu and Özgür Ulusoy 2024), Distillery-SQL (Maamari et al. 2024), Solid-SQL (Liu et al. 2025), TA-SQL (Qu et al. 2024), Reasoning-SQL (Pourreza et al. 2025b), SQL-R1 (Ma et al. 2025); **(ii) backward schema linking**—SQL-to-Schema (Yang et al. 2024), RSL-SQL (Cao et al. 2024); and **(iii) graph-based** approaches—RAT-SQL (Wang et al. 2020), LGESQL (Cao et al. 2021), SADGA (Cai et al. 2021), S²SQL (Hui et al. 2022), ISESSL-SQL (Liu et al. 2022), ShadowGNN (Chen et al. 2021), SchemaGraph-SQL (Safdarian et al. 2025).

Building on the above taxonomy, deploying schema linking at industrial scale exposes three recurring bottlenecks. First, **context limitations**: database-level methods often exceed LLM context windows and inflate computation on large schemas. Second, **inefficiency**: element-level methods require $O(|S|)$ scoring passes, which become impractical as $|S|$ grows. Third, a **recall-noise trade-off**: dual-encoder retrievers achieve high strict recall mainly by returning many candidates, reintroducing irrelevant schema elements and negating token savings.

Method

To address the limitations of traditional schema linking methods in large industrial databases, we draw inspiration from the **exploratory** and **interactive** workflows of human database engineers. Instead of memorizing an unfamiliar database, they typically locate information through multi-step targeted SQL queries and semantic search. Inspired by this pragmatic approach, in this paper, we reframe schema linking as an interactive, sequential decision-making problem. We propose **AutoLink**, a novel schema linking method built around an autonomous agent powered by a Large Language Model (LLM), tasked with dynamically exploring and expanding the linked schema subset (S_{linked}) for given natural language question (Q) **without ever seeing the full database schema** (S_{full}). This process is modeled as the agent, guided by an LLM policy (π), executing a sequence of actions within external environments (\mathcal{E}), with the aim of maximizing the strict recall of ground-truth schema elements based on the interaction trajectory. The overall framework of our proposed method is illustrated in Figure 1.

External Environment \mathcal{E}

To enable agent probing, a pre-built environment (\mathcal{E}) is provided for each unique database, comprising two distinct, complementary components: the *Database Environment* \mathcal{E}_{DB} and the *Schema Vector Store Environment* \mathcal{E}_{VS} .

Env1: Database Environment (\mathcal{E}_{DB}) This environment is the live database itself, providing a direct interface for schema and data exploration via SQL execution. It is defined as a function that maps an input SQL query to a formatted execution result:

$$\mathcal{E}_{\text{DB}} : \text{SQL} \rightarrow R_{\text{SQL}}, \quad (1)$$

the output R_{SQL} is a structured textual response designed for exploratory operations. The R_{SQL} output dynamically provides either execution results (truncated if the number of rows exceeds 5) for successful query, or specific error / timeout messages for failures, enabling clear and actionable agent feedback.

Env2: Schema Vector Store Environment (\mathcal{E}_{VS}) To facilitate efficient semantic search, we construct a vector database of all columns in the schema. For each column $c_i \in S_{\text{full}}$, we create a textual document by concatenating its core metadata: the column name, parent table’s name, data type, and description (if available). We then use a powerful text encoder, i.e., bge-large-en-v1.5 (Chen et al. 2024), to compute a dense vector representation for each column document, which is indexed into a vector store \mathcal{V} .

This environment is designed to bridge the semantic gap between a natural language concept and concrete schema elements. Its function is defined as:

$$\mathcal{E}_{\text{VS}} : (q_{\text{nl}}, K, \mathcal{C}_{\text{excl}}) \xrightarrow{\text{retrieve top-}K \text{ columns}} S_{\text{subset}}, \quad (2)$$

the environment takes a natural language query q_{nl} , a retrieval count K , and a set of already retrieved columns $\mathcal{C}_{\text{excl}}$ to avoid redundancy. It performs an Approximate Nearest Neighbor (ANN) search (Liu et al. 2004) within the vector space of columns not in $\mathcal{C}_{\text{excl}}$. While the search identifies the

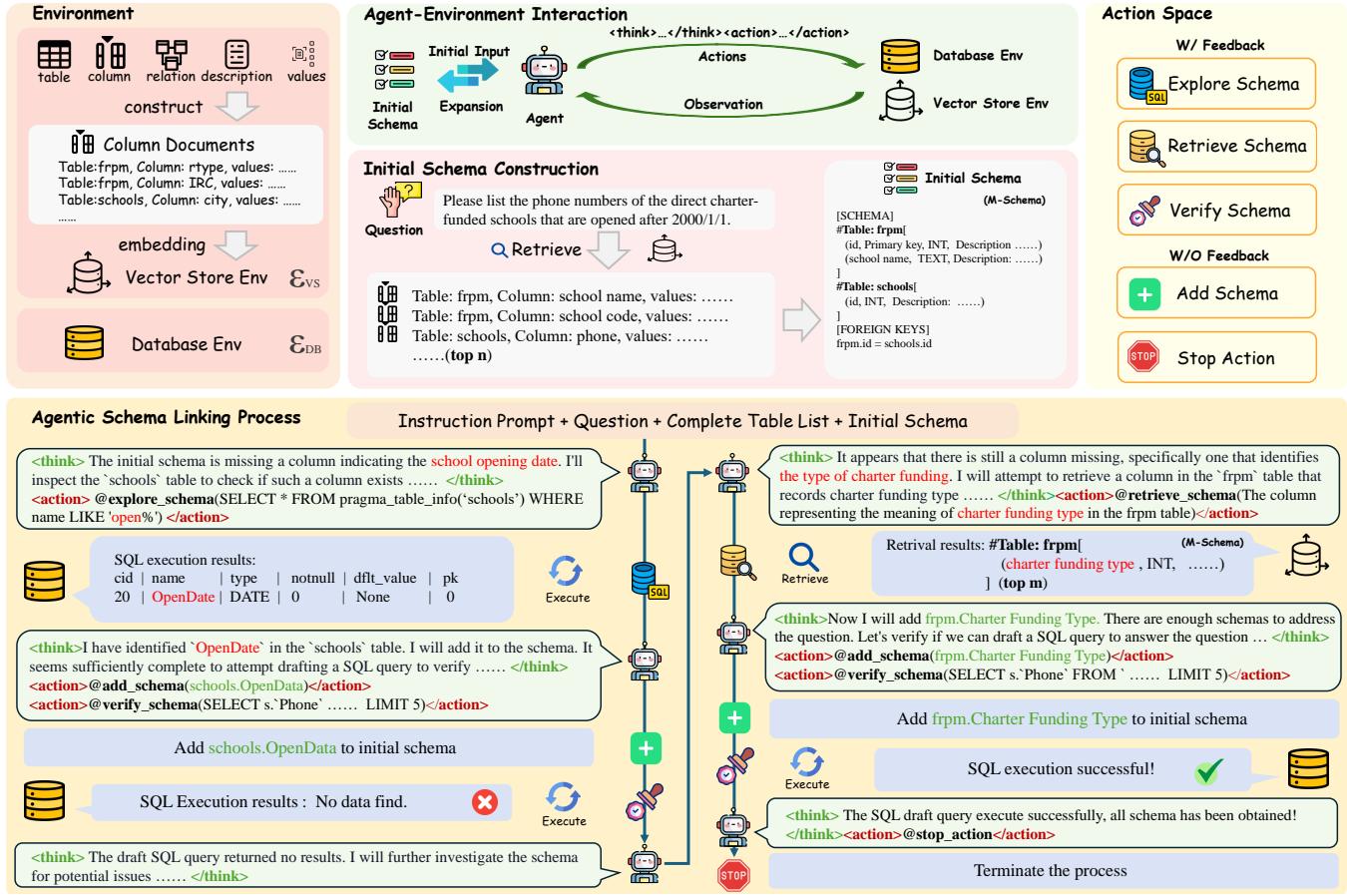


Figure 1: Overview of AutoLink: iterative interactive agentic schema linking framework.

top- K most relevant columns, the output S_{subset} is a **fully structured schema snippet**. This snippet is constructed by gathering all metadata for the retrieved columns, grouping them by their parent tables, and formatting them in a human-readable **M-Schema** style (Gao et al. 2025) that includes data types, keys, and sample values.

We provide **more details on the setup and construction of these two environments** in Appendix Details of Environment Construction.

Agent-Environment Interaction

Our LLM agent, acting as the decision-making policy π , engages the environment \mathcal{E} in a multi-turn dialogue, driven by the interaction history H . This policy is entirely **prompt-based (training-free)** and the complete prompt is provided in Appendix Prompt Templates. The objective is to progressively add potential relevant schema elements for a given user question, obtaining the final linked schema $S_{\text{linked}} \subset S_{\text{full}}$. The process commences with an initial context H_0 . To effectively initiate the multi-turn exploration, the agent’s first-turn input H_0 is meticulously constructed by concatenating four critical components:

1) Instruction Prompt (I) High-level instructions defining the agent’s goal and available actions.

2) User Question (Q) The original user query.

3) Complete Table List (T) All table names in the database (excluding any columns, the total number of tables is typically manageable, e.g., < 100), which provides essential structural context to the agent (the agent needs to know which tables it can query).

4) Initial Schema ($S_{\text{linked}}^{(0)}$) This component is generated by a **single, non-agent-driven call** to the schema vector store environment E_{VS} . We use the original user question Q as query to retrieve an initial set of candidate columns, creating the initial schema $S_{\text{linked}}^{(0)} = E_{\text{VS}}(Q, n, \emptyset)$, where n is a **relatively large hyperparameter** (e.g., 50 or 100) to ensure a broad, albeit potentially incomplete, initial set of schema elements highly relevant to the user’s question. **The trade-off between recall and noise will be examined in the experimental section.** This provides the agent with crucial structural context beyond just table names, facilitating subsequent decision-making. S_{linked} is also initialized directly with $S_{\text{linked}}^{(0)}$, i.e., $S_{\text{linked}} = S_{\text{linked}}^{(0)}$, and will then be updated throughout the interaction.

In each subsequent turn t , the agent generates a reasoning trace θ_t (enclosed within $<\text{think}>/<\text{think}>$ tags) and a set of actions A_t (within $<\text{actions}>/<\text{actions}>$

tags) based on the current history H_t :

$$(\theta_t, A_t) = \pi(H_t). \quad (3)$$

The environment \mathcal{E} then executes these actions, yielding an observation O_t :

$$O_t = \mathcal{E}(A_t). \quad (4)$$

This resulting triplet (θ_t, A_t, O_t) , representing a full turn of interaction, is appended to the history to form H_{t+1} . This loop continues until the agent terminates the process.

Action Space and Agentic Schema Linking

The agent's action space \mathcal{A} provides a set of specialized tools for exploration, verification, and state management. These actions are divided into two categories based on their interaction with the environment.

Actions with Feedback These are the agent's primary tools for gathering new information from the environment.

1. @explore_schema This action aims to interact with the Database Environment (\mathcal{E}_{DB}) by executing exploratory SQL queries, primarily targeting schema metadata or small samples of data, **rather than directly answering the user's question**. For example, the agent can query all columns that contain certain characters (e.g., *id* or *name*) with fuzzy matching, column descriptions and sample column values for certain columns, or examine a table's primary or foreign keys, and other structural metadata, etc.

2. @retrieve_schema This action directly explores missing schema elements within the Schema Vector Store Environment (\mathcal{E}_{VS}). Unlike simple user-question rewrite like CHESS (Talaei et al. 2024), it provides a strong signal for missing schema search. Leveraging the user's natural language question, known table names, and the currently incomplete schema, the agent can directly **infer required missing column names, descriptions or concepts as new query action**. This approach is especially powerful for high-level or ambiguous user queries, narrowing the semantic search space and enabling the discovery of specific, relevant columns beyond simple rephrasing.

Specifically, this action generates a set of retrieved candidate schema elements as: $S_{retrieved} = \mathcal{E}_{VS}(q, m, \mathcal{C}_{excl})$, where **m is a relatively small number** (e.g., 3 or 5) since this action is a targeted retrieval. Previously retrieved columns (from the initial schema or prior @retrieve_schema calls) are excluded from the search space.

3. @verify_schema This action interacts with \mathcal{E}_{DB} and functions as a holistic hypothesis test by attempting to **execute a full SQL query specifically designed to answer the user's question**. The primary purpose is **not to generate the final query result**, but to check for schema sufficiency. A successful execution may indicate completeness, while an error provides a strong, direct signal about which specific schema elements are still missing.

Actions without Feedback These actions manage the agent's internal state and control the workflow.

4. @add_schema This action serves as the agent's mechanism for explicitly committing newly discovered relevant schema elements and updating the linked schema S_{linked} ,

which we call **Schema Expansion**. After actions with feedback yields new, relevant schema elements, the agent can adopt this action. To improve schema linking recall, we particularly encourage the agent to add schema returned by the @retrieve_schema action.

The agent's output for this action specifies the schema elements to be added, presented as a semicolon-separated list of *table.name.column.name* strings within the *add_schema()*. Upon execution, the system processes these identifiers by collecting their complete meta-information. Let S_{added} denote the set of these fully described schema elements. S_{linked} is then updated by merging these newly added schema elements:

$$S_{linked} \leftarrow S_{linked} \cup S_{added}. \quad (5)$$

It is important to note that while the agent can output one or more actions per turn, @add_schema cannot be the sole action, since outputting it alone would leave the agent without environmental feedback for subsequent turns. Therefore, the instruction prompt explicitly requires @add_schema to always be paired with at least one feedback-providing action or @stop_action.

5. @stop_action This action terminates the multi-turn interaction process. The agent uses this action when, based on its analysis of the dialogue history (including initial schema and expanded schema elements through previous @add_schema actions), it determines that enough information has been gathered to answer the user's question. This decision primarily relies on the agent's contextual understanding and significantly influenced by the outcomes of the @verify_schema action. Additionally, the process is also terminated if the number of interaction turns exceeds a pre-defined maximum (**10 turns** in this paper), serving as a safeguard against endless loops.

SQL Generation

With the final linked schema S_{linked} obtained through our iterative, agent-driven process, the subsequent step is **SQL generation**. It is important to note that SQL generation itself is **not the core focus** of this paper, and we leverage existing techniques for this phase. We employ an existing LLM as the SQL generation policy (π_{SQL}). Specifically, given the user's original natural language question Q and final linked schema S_{linked} , the policy adopts a self-consistency strategy (Wang et al. 2023) to sample multiple SQL candidates, then performs syntactic correction (Talaei et al. 2024; Pourreza et al. 2025a) and majority voting (Deng et al. 2025) to derive the final SQL statement. Please refer to the Appendix Detail of SQL Generation.

Experiment

Implementation Details

Experimental Setup Considering the cost-effectiveness of the DeepSeek series model, DeepSeek-V3 serves as the LLM policy π for schema linking (AutoLink), and DeepSeek-R1 and DeepSeek-V3 are employed as the policy π_{SQL} for the subsequent SQL generation phase. The reasoning LLM (DeepSeek-R1) is not chosen for AutoLink's policy due to observed **instruction following degradation** (Fu

Method	Full Input	Bird Dev				Spider2.0-Lite					
		SRR \uparrow	\bar{C}^{\downarrow}	Avg. Tokens \downarrow		SRR $_{all}^{\uparrow}$	SRR $_{bq}^{\uparrow}$	SRR $_{sf}^{\uparrow}$	SRR $_{local}^{\uparrow}$	\bar{C}^{\downarrow}	Avg. Tokens \downarrow
DE-SL (BGE-Large)	✓	35.5	35.7	–	43.6	28.2	57.1	87.5	153.8	–	–
CE-SL (BGE-reranker)	✓	72.4	35.7	–	57.6	42.3	72.6	95.8	153.8	–	–
MCS-SQL	✓	85.7	7.5	29.8K	58.9	57.8	54.8	79.2	45.1	168.9K	
SQL-to-Schema	✓	92.5	13.5	19.4K	64.0	64.8	54.8	91.7	49.0	171.9K	
CHESS	✓	89.7	4.5	–	–	–	–	–	–	–	–
RSL-SQL	✓	93.3	13.0	14.8K	52.0	52.8	44.1	75.0	25.8	29.2K	
LinkAlign	✗	–	–	–	36.4	22.5	51.2	66.7	21.1	66.7K	
AutoLink (ours)	✗	97.4	35.8	8.0K	91.2	93.7	85.7	95.8	159.4	21.2K	

Table 1: **Performance comparison of schema linking on Bird Dev and Spider 2.0-Lite.** We report the overall Strict Recall Rate (SRR $_{all}$) for both datasets. For Spider 2.0-Lite, SRR is further broken down by SQL dialect: BigQuery (SRR $_{bq}$), Snowflake (SRR $_{sf}$), and SQLite (SRR $_{local}$). \bar{C} denotes the average number of columns included in the simplified schema (S_{linked}) after schema linking. **Full Input** indicates whether the entire database schema is provided to the LLM or if all database schemas are iterated through.

et al. 2025), which occasionally leads to deviations from the required thought and action format. The **key hyperparameters** for our AutoLink framework include top- n for initial schema retrieval (varied between 5 and 100 during experiments); top- m for the `@retrieve_schema` action, fixed at 3; and a maximum interaction turn limit, set to 10.

Datasets Our evaluation is conducted on two distinct Text-to-SQL benchmarks: the Spider 2.0-Lite dataset (Lei et al. 2025) and the Bird-Dev dataset (Li et al. 2023c). Bird-Dev comprises 11 databases, featuring an average of 80 columns per database, and includes 1,543 complex SQL query use cases. Spider 2.0-Lite, derived from industrial applications, is designed to reflect the scale of real-world databases. It presents a greater challenge than Bird-Dev, with its databases averaging over 800 columns, more intricate SQL queries, and support for multiple SQL dialects (including BigQuery, Snowflake, and SQLite). This dataset contains 547 test cases.

Evaluation Metrics Model performance is evaluated using 3 primary metrics. The **Strict Recall Rate (SRR)** (Cao et al. 2024) measures the effectiveness of schema linking, defined as the proportion of test cases where the final simplified schema fully contains all required gold schemas. For SQL generation, we report **Execution Accuracy (EX)**, consistent with the official definitions of the Spider and BIRD benchmarks, which measures the consistency between the execution results of predicted and gold SQL queries. Lastly, for LLM-based methods, we report **Avg. Token Consumption**, representing the average total tokens (input plus output) per example. This metric serves as a key metric of computational cost, as overall inference latency is highly susceptible to external factors (e.g., API, network variability), and the time spent on SQL execution within our method is negligible compared to LLM decoding.

Baselines To evaluate AutoLink’s performance, we compare it against a diverse set of established methods across both schema linking and SQL generation tasks. For element-level schema linking, our baselines include **DE-SL**, which employs a dual-encoder (BGE-Large-v1.5) to pre-cache

column documents and retrieve the top- K most similar columns, and **CE-SL**, which uses a cross-encoder (BGE-reranker) to compute and rank the similarity between the user query and each column individually for top- K retrieval. Additionally, **CHESS** (Talaei et al. 2024) is included, an LLM-based method that scores the user query against each column, followed by sequential table and column filtering. For database-level schema linking, we consider **MCS-SQL** (Lee et al. 2025), which leverages an LLM to directly output relevant schema elements from the full schema and user query, utilizing 5-time sampling decoding to merge results. Similarly, **SQL-to-schema** (Yang et al. 2024) uses an LLM to generate an SQL statement from which involved tables and columns are parsed, also employing 5-time sampling decoding. **RSL-SQL** (Cao et al. 2024) is a hybrid approach combining elements of MCS-SQL and SQL-to-schema, typically with a single decoding pass. Furthermore, **LinkAlign** (Wang, Liu, and Yang 2025) stands as a baseline for its query rewriting and multi-agent discussion framework that also bypasses the need for schema element retrieval.

Main Results of Schema Linking

As shown in Table 1, all models (except for **DE-SL** and **CE-SL**) are implemented using DeepSeek-V3 as the backbone to ensure fair and consistent comparison. For the hyperparameter top- n for initial schema retrieval, we set 30 on Bird Dev and 100 on Spider 2.0-Lite. The experimental results demonstrate that **AutoLink** achieves a significant advantage over baseline methods in terms of SRR and average token consumption across both the Bird and Spider 2.0-Lite. On the more challenging Spider 2.0-Lite benchmark, our method achieves a **27.2%** improvement in strict recall rate compared to the second place (**SQL-to-Schema**), while reducing the maximum token consumption by **87.7%**. Compared with encoder-based methods (**DE-SL** and **CE-SL**), although the number of recalled columns is close, our method improves SRR by an average of approximately **40%**.

Compared to methods requiring full schema input, such as **MCS-SQL**, **SQL-to-Schema**, and **RSL-SQL**, we ob-

Method	Model	EX [†]	Avg. Tokens [↓]
Spider-Agent	QwQ	11.33	—
	GPT-4o	13.16	—
	DeepSeek-R1	13.71	—
	o1-preview	23.03	—
	o3-mini	23.40	—
	Claude-3.7-Sonnet	28.52	—
CHESS	GPT-4o	3.84	—
LinkAlign	DeepSeek-R1	33.09	—
RSL-SQL [†]	DeepSeek-R1	30.53	50.0K
ReFORCE [†]	DeepSeek-R1	29.62	81.1K
ReFORCE	o1-preview	30.35	81.1K
ReFORCE	GPT-o3	37.84	81.1K
AutoLink	DeepSeek-R1	<u>34.92</u>	38.0K

Table 2: Execution Accuracy (EX) comparison of different methods on the Spider 2.0-Lite dataset. [†] indicates results obtained through independent reproduction. Avg. Tokens indicates the average number of tokens consumed to generate a SQL.

serve that while they perform acceptably on smaller datasets like Bird, their performance sharply declines on larger, more complex databases like Spider 2.0-Lite. This is due to the long context lengths and high resource consumption at scale. Although these methods can theoretically enhance SRR by increasing sampling decoding iterations, we found that the SRR growth becomes negligible and quickly saturates, particularly on the Spider 2.0-Lite. *Blindly increasing decoding attempts yields minimal performance gains*, as the results across different decoding passes often exhibit low diversity. Consequently, **these methods struggle to effectively control the number of recalled columns to achieve a comparable scale to ours**. In contrast, our method consistently maintains strong performance across varying dataset scales. We demonstrate the scatter plot of recalled columns versus SRR in Appendix More Experiment Results, and our approach still **achieves superior recall rates even when recalling a similar small number of columns**. Regarding the comparison of Token consumption, we found that compared with above methods, our method significantly reduces the Token overhead. Compared with RSL-SQL, although its Token overhead is not large, its recall performance in large-scale databases is relatively low.

In contrast to methods like **CHESS** and **LinkAlign**, which **prioritize noise reduction by aiming to include only the columns strictly required by the gold SQL**, our approach differs. This noise-minimization strategy carries substantial risk, as evidenced by LinkAlign’s mere 36.4% SRR, despite its minimal average of 21.1 recalled columns. Such a low recall rate severely compromises the utility of the simplified schema for subsequent SQL generation.

Results of SQL Generation

The comparative results in Table 2 and Table 3 demonstrate that **AutoLink** achieves competitive EX on both Spi-

Method	Model	EX [†]	Avg. Tokens [↓]
MAC-SQL	GPT-4	59.39	6.8K
	TA-SQL	56.19	7.3K
	RSL-SQL	67.21	7.5K
	CHESS	<u>68.31</u>	14.5K
AutoLink	DeepSeek-v3	66.36	8.0K
AutoLink	Gemini-1.5-Pro	68.71	8.0K

Method	SRR _{n=5}	SRR _{n=50}	SRR _{n=100}
AutoLink	79.2	88.0	91.2
- w/o Verify Schema	77.2 _{±2.0}	86.8 _{±1.2}	89.6 _{±1.6}
- w/o Explore Schema	76.8 _{±2.4}	84.8 _{±3.2}	88.8 _{±2.4}
- w/o Retrieve Schema	72.4 _{±6.8}	80.4 _{±7.6}	84.5 _{±6.7}

Table 3: Execution Accuracy (EX) comparison of different methods the Bird Dev.

Method	SRR _{n=5}	SRR _{n=50}	SRR _{n=100}
AutoLink	79.2	88.0	91.2
- w/o Verify Schema	77.2 _{±2.0}	86.8 _{±1.2}	89.6 _{±1.6}
- w/o Explore Schema	76.8 _{±2.4}	84.8 _{±3.2}	88.8 _{±2.4}
- w/o Retrieve Schema	72.4 _{±6.8}	80.4 _{±7.6}	84.5 _{±6.7}

Table 4: Ablation study on the impact of different schema linking actions under varying numbers of initial candidates n on Spider 2.0-Lite.

der 2.0-Lite and Bird. On Spider 2.0-Lite, AutoLink attains an EX score of **34.92%** using DeepSeek-R1, which outperforms most baseline approaches and is highly competitive with the best-performing method, ReFoRCE (37.84% with GPT-o3). Notably, ReFoRCE generates eight candidate SQL queries per example, whereas AutoLink generates only five. This difference results in a significant advantage for AutoLink in terms of token consumption—AutoLink requires only **38.0K** tokens on average, less than half of ReFoRCE’s 81.1K. When evaluated using the same model, DeepSeek-R1, our method achieves the best performance. Similarly, on the Bird dev set, AutoLink also achieves strong results. With Gemini-1.5-Pro as the backbone model, AutoLink achieves an EX score of **68.71%**, which is competitive with or superior to strong baselines such as **CHESS** and **RSL-SQL**.

Ablation Study

The ablation results in Table 4 demonstrate the importance of each core agent action for schema linking under initial retrieval sizes of top-5, top-50, and top-100. Removing any single action—**@retrieve_schema**, **@explore_schema**, or **@verify_schema**—leads to a clear drop in SRR across all initial top- n settings. Notably, removing schema retrieval has the largest impact, highlighting its key role in identifying relevant columns. Without this capability, the agent is much less able to anchor its reasoning in the most promising schema elements. Excluding database exploration also consistently reduces performance, underscoring its value for resolving ambiguities when facing unfamiliar databases. Although omitting verification results in a smaller decrease in SRR, its contribution is evident across all candidate sizes, reflecting the importance of continual self-assessment within the agent’s reasoning process.

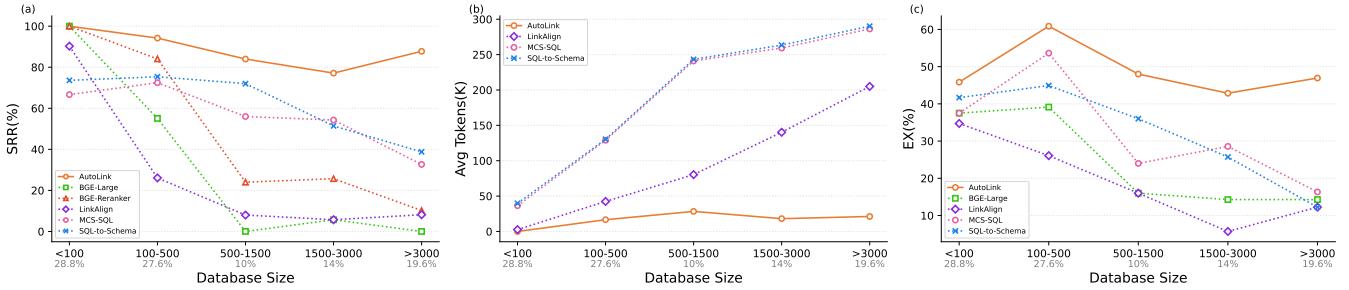


Figure 2: Scalability comparison across databases on Spider 2.0-Lite of varying sizes in terms of (a) Strict Recall Rate ($SRR \uparrow$), (b) Average Tokens Consumption ($Avg. Tokens \downarrow$), and (c) Execution Accuracy ($EX \uparrow$). Percentages below each bin indicate the proportion of databases within each size range.

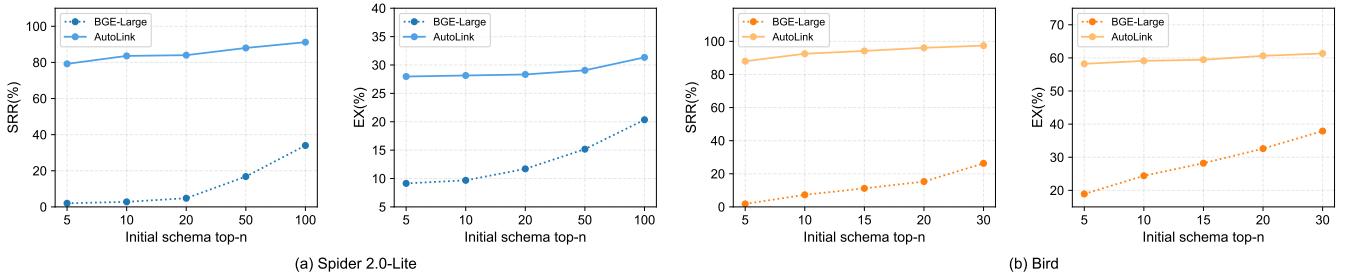


Figure 3: SRR and EX comparison of schema linking at different initial top-n on Spider 2.0-Lite and Bird dev set.

Scalability Analysis of Different Database Scales

Figure 2 compares the SRR of various schema linking methods: **AutoLink**, **BGE-Large (DE-SL)**, **BGE-reranker (CE-SL)**, and **LinkAlign**, etc., on Spider 2.0-Lite across increasing database scales. All methods exhibit a significant decline in SRR as the database size grows, though the decrease of **AutoLink** is notably slower. In large databases exceeding 3,000 columns, the SRR of all baseline models drops below **40%**, yet AutoLink’s SRR remains near **90%**. For token consumption ((b) in Figure 2), all methods generally show an upward trend. AutoLink consistently demonstrates the lowest average token usage across all database sizes. This efficiency stems from AutoLink’s iterative approach, which begins with a smaller schema and gradually expands it, resulting to minimal changes in token consumption across different database scales.

To ensure a fair comparison, our SQL generation method is applied to the linked schemas produced by different schema linking approaches. Regarding EX ((c) in Figure 2), the performance of all baselines declines as schema size increases. Crucially, we observe a direct correlation: **models with higher SRR tend to exhibit higher EX**. This highlights that a higher SRR is instrumental in improving the EX of subsequent SQL generation. This is intuitively sound because if the SQL generator (i.e., the LLM), does not hallucinate, it will only utilize the schema elements provided by the schema linking step. Consequently, an incomplete schema (due to low SRR) means the generated SQL may lack necessary tables and columns from the gold SQL, making it highly probable that the generated query will be incor-

rect. **AutoLink** consistently achieves the highest EX across all size ranges, significantly outperforming baselines on the largest databases due to its superior SRR.

Analysis of Hyperparameter

As shown in Figure 3, **AutoLink** consistently outperforms BGE Large in SRR under various initial top-n schema retrieval settings. Increasing top-n improves SRR for both methods, but AutoLink’s agent-driven exploration gives it a notable advantage even at **small top-n values** (e.g., top-5) by effectively identifying and incorporating missing schema components. This robustness holds across datasets of different scales, demonstrating good scalability.

For downstream SQL EX on Spider 2.0-Lite, AutoLink consistently achieves higher accuracy than BGE across all top-n settings. While its performance rises with larger top-n, the improvement plateaus at high values, suggesting that **most critical schema elements can already be recovered with smaller candidate sets**. The advantage is most pronounced at low top-n, highlighting AutoLink’s effectiveness when initial schema retrieval is incomplete. Additional hyperparameter analysis, including the effects of max turn and top-m expansion, is provided in the Appendix More Experiment Results.

Conclusion

In this paper, we propose AutoLink, a novel framework that redefines schema linking as an adaptive, agent-driven process. By orchestrating semantic retrieval from a vector store and utilizing lightweight SQL probes, our LLM-

powered agent iteratively and autonomously assembles only the schema elements truly necessary for a given query, without requiring the full database schema as input. This unified mechanism achieves state-of-the-art strict recall on Spider 2.0-Lite and Bird, significantly reduces token usage, and demonstrates exceptional scalability and robustness on large schemas with thousands of columns, thereby providing a practical and efficient foundation for industrial-scale Text-to-SQL systems.

Acknowledgments

This research was supported by *National Natural Science Foundation of China* (No.62406121) and *National Science Foundation of Hubei Province, China* (No.2024AFB189).

References

- Caferoglu, H. A.; and Özgür Ulusoy. 2024. E-SQL: Direct Schema Linking via Question Enrichment in Text-to-SQL. arXiv:2409.16751.
- Cai, R.; Yuan, J.; Xu, B.; and Hao, Z. 2021. Sadga: Structure-aware dual graph aggregation network for text-to-sql. *Advances in Neural Information Processing Systems*, 34: 7664–7676.
- Cao, R.; Chen, L.; Chen, Z.; Zhao, Y.; Zhu, S.; and Yu, K. 2021. LGESQL: Line Graph Enhanced Text-to-SQL Model with Mixed Local and Non-Local Relations. In Zong, C.; Xia, F.; Li, W.; and Navigli, R., eds., *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2541–2555. Online: Association for Computational Linguistics.
- Cao, Z.; Zheng, Y.; Fan, Z.; Zhang, X.; Chen, W.; and Bai, X. 2024. RSL-SQL: Robust Schema Linking in Text-to-SQL Generation. arXiv:2411.00073.
- Chen, J.; Xiao, S.; Zhang, P.; Luo, K.; Lian, D.; and Liu, Z. 2024. M3-Embedding: Multi-Linguality, Multi-Functionality, Multi-Granularity Text Embeddings Through Self-Knowledge Distillation. In Ku, L.-W.; Martins, A.; and Srikumar, V., eds., *Findings of the Association for Computational Linguistics: ACL 2024*, 2318–2335. Bangkok, Thailand: Association for Computational Linguistics.
- Chen, Z.; Chen, L.; Zhao, Y.; Cao, R.; Xu, Z.; Zhu, S.; and Yu, K. 2021. ShadowGNN: Graph Projection Neural Network for Text-to-SQL Parser. In Toutanova, K.; Rumshisky, A.; Zettlemoyer, L.; Hakkani-Tur, D.; Beltagy, I.; Bethard, S.; Cotterell, R.; Chakraborty, T.; and Zhou, Y., eds., *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 5567–5577. Online: Association for Computational Linguistics.
- Deng, M.; Ramachandran, A.; Xu, C.; Hu, L.; Yao, Z.; Datta, A.; and Zhang, H. 2025. ReFoRCE: A Text-to-SQL Agent with Self-Refinement, Format Restriction, and Column Exploration. In *ICLR 2025 Workshop: VerifAI: AI Verification in the Wild*.
- Devlin, J.; Chang, M.-W.; Lee, K.; and Toutanova, K. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 4171–4186.
- Dong, X.; Zhang, C.; Ge, Y.; Mao, Y.; Gao, Y.; lu Chen; Lin, J.; and Lou, D. 2023. C3: Zero-shot Text-to-SQL with ChatGPT. arXiv:2307.07306.
- Fu, T.; Gu, J.; Li, Y.; Qu, X.; and Cheng, Y. 2025. Scaling Reasoning, Losing Control: Evaluating Instruction Following in Large Reasoning Models. arXiv:2505.14810.
- Gao, D.; Wang, H.; Li, Y.; Sun, X.; Qian, Y.; Ding, B.; and Zhou, J. 2024. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. *Proc. VLDB Endow.*, 17(5): 1132–1145.
- Gao, Y.; Liu, Y.; Li, X.; Shi, X.; Zhu, Y.; Wang, Y.; Li, S.; Li, W.; Hong, Y.; Luo, Z.; Gao, J.; Mou, L.; and Li, Y. 2025. A Preview of XiYan-SQL: A Multi-Generator Ensemble Framework for Text-to-SQL. arXiv:2411.08599.
- Hong, Z.; Yuan, Z.; Zhang, Q.; Chen, H.; Dong, J.; Huang, F.; and Huang, X. 2025. Next-Generation Database Interfaces: A Survey of LLM-Based Text-to-SQL . *IEEE Transactions on Knowledge & Data Engineering*, 37(12): 7328–7345.
- Hui, B.; Geng, R.; Wang, L.; Qin, B.; Li, Y.; Li, B.; Sun, J.; and Li, Y. 2022. S²SQL: Injecting Syntax to Question-Schema Interaction Graph Encoder for Text-to-SQL Parsers. In Muresan, S.; Nakov, P.; and Villavicencio, A., eds., *Findings of the Association for Computational Linguistics: ACL 2022*, 1254–1262. Dublin, Ireland: Association for Computational Linguistics.
- Johnson, J.; Douze, M.; and Jégou, H. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3): 535–547.
- Katsogiannis-Meimarakis, G.; and Koutrika, G. 2023. A survey on deep learning approaches for text-to-SQL. *The VLDB Journal*, 32(4): 905–936.
- Lee, D.; Park, C.; Kim, J.; and Park, H. 2025. MCS-SQL: Leveraging Multiple Prompts and Multiple-Choice Selection For Text-to-SQL Generation. In Rambow, O.; Waner, L.; Apidianaki, M.; Al-Khalifa, H.; Eugenio, B. D.; and Schockaert, S., eds., *Proceedings of the 31st International Conference on Computational Linguistics*, 337–353. Abu Dhabi, UAE: Association for Computational Linguistics.
- Lei, F.; Chen, J.; Ye, Y.; Cao, R.; Shin, D.; SU, H.; SUO, Z.; Gao, H.; Hu, W.; Yin, P.; Zhong, V.; Xiong, C.; Sun, R.; Liu, Q.; Wang, S.; and Yu, T. 2025. Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows. In *The Thirteenth International Conference on Learning Representations*.
- Li, B.; Luo, Y.; Chai, C.; Li, G.; and Tang, N. 2024a. The Dawn of Natural Language to SQL: Are We Fully Ready? *Proc. VLDB Endow.*, 17(11): 3318–3331.

- Li, H.; Zhang, J.; Li, C.; and Chen, H. 2023a. RESDSQL: decoupling schema linking and skeleton parsing for text-to-SQL. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence, AAAI'23/IAAI'23/EAAI'23*. AAAI Press. ISBN 978-1-57735-880-0.
- Li, H.; Zhang, J.; Liu, H.; Fan, J.; Zhang, X.; Zhu, J.; Wei, R.; Pan, H.; Li, C.; and Chen, H. 2024b. CodeS: Towards Building Open-source Language Models for Text-to-SQL. *Proc. ACM Manag. Data*, 2(3).
- Li, J.; Hui, B.; Cheng, R.; Qin, B.; Ma, C.; Huo, N.; Huang, F.; Du, W.; Si, L.; and Li, Y. 2023b. Graphix-T5: mixing pre-trained transformers with graph-aware layers for text-to-SQL parsing. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence, AAAI'23/IAAI'23/EAAI'23*. AAAI Press. ISBN 978-1-57735-880-0.
- Li, J.; Hui, B.; Qu, G.; Yang, J.; Li, B.; Li, B.; Wang, B.; Qin, B.; Geng, R.; Huo, N.; Zhou, X.; Ma, C.; Li, G.; Chang, K. C.; Huang, F.; Cheng, R.; and Li, Y. 2023c. Can LLM already serve as a database interface? a big bench for large-scale database grounded text-to-SQLs. In *Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS '23*. Red Hook, NY, USA: Curran Associates Inc.
- Liu, A.; Hu, X.; Lin, L.; and Wen, L. 2022. Semantic Enhanced Text-to-SQL Parsing via Iteratively Learning Schema Linking Graph. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD '22*, 1021–1030. New York, NY, USA: Association for Computing Machinery. ISBN 9781450393850.
- Liu, G.; Tan, Y.; Zhong, R.; Xie, Y.; Zhao, L.; Wang, Q.; Hu, B.; and Li, Z. 2025. Solid-SQL: Enhanced Schema-linking based In-context Learning for Robust Text-to-SQL. In Rambow, O.; Wanner, L.; Apidianaki, M.; Al-Khalifa, H.; Eugenio, B. D.; and Schockaert, S., eds., *Proceedings of the 31st International Conference on Computational Linguistics*, 9793–9803. Abu Dhabi, UAE: Association for Computational Linguistics.
- Liu, T.; Moore, A.; Yang, K.; and Gray, A. 2004. An Investigation of Practical Approximate Nearest Neighbor Algorithms. In Saul, L.; Weiss, Y.; and Bottou, L., eds., *Advances in Neural Information Processing Systems*, volume 17. MIT Press.
- Ma, P.; Zhuang, X.; Xu, C.; Jiang, X.; Chen, R.; and Guo, J. 2025. SQL-R1: Training Natural Language to SQL Reasoning Model By Reinforcement Learning. arXiv:2504.08600.
- Maamari, K.; Abubaker, F.; Jaroslawicz, D.; and Mhedhbi, A. 2024. The Death of Schema Linking? Text-to-SQL in the Age of Well-Reasoned Language Models. In *NeurIPS 2024 Third Table Representation Learning Workshop*.
- Pourreza, M.; Li, H.; Sun, R.; Chung, Y.; Talaei, S.; Kakkar, G. T.; Gan, Y.; Saberi, A.; Ozcan, F.; and Arik, S. 2025a. CHASE-SQL: Multi-Path Reasoning and Preference Optimized Candidate Selection in Text-to-SQL. In Yue, Y.; Garg, A.; Peng, N.; Sha, F.; and Yu, R., eds., *International Conference on Representation Learning*, volume 2025, 60385–60415.
- Pourreza, M.; and Rafiei, D. 2023. DIN-SQL: decomposed in-context learning of text-to-SQL with self-correction. In *Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS '23*. Red Hook, NY, USA: Curran Associates Inc.
- Pourreza, M.; Talaei, S.; Sun, R.; Wan, X.; Li, H.; Mirhoseini, A.; Saberi, A.; and Arik, S. O. 2025b. Reasoning-SQL: Reinforcement Learning with SQL Tailored Partial Rewards for Reasoning-Enhanced Text-to-SQL. arXiv:2503.23157.
- Qu, G.; Li, J.; Li, B.; Qin, B.; Huo, N.; Ma, C.; and Cheng, R. 2024. Before Generation, Align it! A Novel and Effective Strategy for Mitigating Hallucinations in Text-to-SQL Generation. In Ku, L.-W.; Martins, A.; and Srikumar, V., eds., *Findings of the Association for Computational Linguistics ACL 2024*, 5456–5471. Bangkok, Thailand and virtual meeting: Association for Computational Linguistics.
- Safdarian, A.; Mohammadi, M.; Jahanbakhsh, E.; Naderi, M. S.; and Faili, H. 2025. SchemaGraphSQL: Efficient Schema Linking with Pathfinding Graph Algorithms for Text-to-SQL on Large-Scale Databases. arXiv:2505.18363.
- Shi, L.; Tang, Z.; Zhang, N.; Zhang, X.; and Yang, Z. 2025. A Survey on Employing Large Language Models for Text-to-SQL Tasks. *ACM Comput. Surv.*, 58(2).
- Talaei, S.; Pourreza, M.; Chang, Y.-C.; Mirhoseini, A.; and Saberi, A. 2024. CHESS: Contextual Harnessing for Efficient SQL Synthesis. arXiv:2405.16755.
- Wang, B.; Ren, C.; Yang, J.; Liang, X.; Bai, J.; Chai, L.; Yan, Z.; Zhang, Q.-W.; Yin, D.; Sun, X.; and Li, Z. 2025. MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL. In Rambow, O.; Wanner, L.; Apidianaki, M.; Al-Khalifa, H.; Eugenio, B. D.; and Schockaert, S., eds., *Proceedings of the 31st International Conference on Computational Linguistics*, 540–557. Abu Dhabi, UAE: Association for Computational Linguistics.
- Wang, B.; Shin, R.; Liu, X.; Polozov, O.; and Richardson, M. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In Jurafsky, D.; Chai, J.; Schluter, N.; and Tetreault, J., eds., *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 7567–7578. Online: Association for Computational Linguistics.
- Wang, X.; Wei, J.; Schuurmans, D.; Le, Q. V.; Chi, E. H.; Narang, S.; Chowdhery, A.; and Zhou, D. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *The Eleventh International Conference on Learning Representations*.
- Wang, Y.; Liu, P.; and Yang, X. 2025. LinkAlign: Scalable Schema Linking for Real-World Large-Scale Multi-Database Text-to-SQL. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Pro-*

cessing, 977–991. Suzhou, China: Association for Computational Linguistics. ISBN 979-8-89176-332-6.

Yang, S.; Su, Q.; Li, Z.; Li, Z.; Mao, H.; Liu, C.; and Zhao, R. 2024. SQL-to-Schema Enhances Schema Linking in Text-to-SQL. In Strauss, C.; Amagasa, T.; Manco, G.; Kotsis, G.; Tjoa, A. M.; and Khalil, I., eds., *Database and Expert Systems Applications*, 139–145. Cham: Springer Nature Switzerland. ISBN 978-3-031-68309-1.

Yu, T.; Zhang, R.; Yang, K.; Yasunaga, M.; Wang, D.; Li, Z.; Ma, J.; Li, I.; Yao, Q.; Roman, S.; Zhang, Z.; and Radev, D. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In Riloff, E.; Chiang, D.; Hockenmaier, J.; and Tsujii, J., eds., *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 3911–3921. Brussels, Belgium: Association for Computational Linguistics.

Details of Environment Construction

This section provides a detailed exposition of the construction and operational principles behind the two core external environments within our **AutoLink** framework: the **Database Environment** (\mathcal{E}_{DB}) and the **Schema Vector Store Environment** (\mathcal{E}_{VS}). These environments are designed to furnish the autonomous agent with the necessary infrastructure for iterative exploration, schema verification, and efficient semantic retrieval.

Database Environment (\mathcal{E}_{DB}) The output R_{SQL} from \mathcal{E}_{DB} , generated for any arbitrary SQL query, is meticulously structured to offer clear and actionable feedback to the agent. This environment is designed for exploratory and experimental SQL queries, thus discouraging overly complex or long-running operations. To ensure efficient interaction and manage the LLM's context length, strict constraints are enforced: 1) The data payload in R_{SQL} is truncated to a maximum of **5 rows**, and 2) Each SQL query's execution is capped at **120 seconds**, with automatic termination if this limit is exceeded. The structure of R_{SQL} adapts to the query's execution outcome, providing detailed feedback for successful operations (non-empty or empty result set) and clear error messages for failures or timeouts. Specific examples are shown below.

- **Successful Query Execution: Non-Empty Result Set**

When a query executes successfully and returns results within the time limit, R_{SQL} provides information structured as follows:

```

1 [Total rows: 123, Execution time: 0.05s,
  ↳ Top-5 rows are shown below]
2 Column1 | Column2
3 -----|-----
4 Value1 | ValueA
5 Value2 | ValueB
6 Value3 | ValueC
7 Value4 | ValueD
8 Value5 | ValueE
9 118 rows truncated ...

```

- **Successful Query Execution: Empty Result Set** If a query executes successfully within the time limit but yields no rows (e.g., a SELECT query with no matches), R_{SQL} provides a distinct message:

```

1 [No data found for the specified query,
  ↳ Execution time: 0.2s]

```

- **SQL Execution Timeout** If a query exceeds the execution time limit, R_{SQL} returns a specific error message indicating the timeout:

```

1 [[ERROR: SQL execution timed out after
  ↳ 120 seconds]]

```

Algorithm 1: AutoLink: Autonomous Schema Exploration and Expansion

Require: Database Environment \mathcal{E}_{DB} , Schema Vector Store Environment \mathcal{E}_{VS}
Require: Instruction Prompt I , User Question Q , Full Database Schema S_{full} , All Table Names T extracted from S_{full}
Require: Maximum Interaction Turns T_{max} , Initial Retrieval Count n , Targeted Retrieval Count m
Ensure: Final Linked Schema S_{linked}

- 1: **Initialize:**
- 2: History $H \leftarrow$ empty string
- 3: Current Linked Schema $S_{linked} \leftarrow \emptyset$
- 4: Excluded Columns $C_{excl} \leftarrow \emptyset$
- 5: **Step 1: Initial Schema Generation**
- 6: $S_{initial} \leftarrow \mathcal{E}_{VS}(Q, n, C_{excl})$ {Retrieve top- n columns from }
- 7: $S_{linked} \leftarrow S_{initial}$
- 8: $C_{excl} \leftarrow$ columns in $S_{initial}$
- 9: $H_0 \leftarrow$ Construct initial context with $I, Q, T, S_{initial}$
- 10: **Step 2: Agent-Environment Interaction**
- 11: **for** $t = 1$ to T_{max} **do**
- 12: Construct current prompt $P_t \leftarrow H_{t-1}$
- 13: $(\theta_t, A_t) \leftarrow \pi(P_t)$ {LLM agent generates reasoning and actions}
- 14: $O_t \leftarrow$ empty string {Initialize observations for current turn}
- 15: stop_flag \leftarrow FALSE
- 16: **for** each action $a \in A_t$ **do**
- 17: **if** $a = @explore_schema(sql_query)$ **then**
- 18: $R_{SQL} \leftarrow \mathcal{E}_{DB}(sql_query)$
- 19: Append R_{SQL} to O_t
- 20: **else if** $a = @retrieve_schema(nl_query)$ **then**
- 21: $S_{retrieved} \leftarrow \mathcal{E}_{VS}(nl_query, m, C_{excl})$
- 22: Append $S_{retrieved}$ to O_t
- 23: **else if** $a = @verify_schema(sql_query)$ **then**
- 24: $R_{SQL} \leftarrow \mathcal{E}_{DB}(sql_query)$
- 25: Append R_{SQL} to O_t
- 26: **else if** $a = @add_schema(schemas)$ **then**
- 27: $S_{added} \leftarrow$ extract full metadata of schemas
- 28: $S_{linked} \leftarrow S_{linked} \cup S_{added}$
- 29: $C_{excl} \leftarrow C_{excl} \cup$ columns in S_{added}
- 30: **else if** $a = @stop_action()$ **then**
- 31: stop_flag \leftarrow TRUE
- 32: **end if**
- 33: **end for**
- 34: $H_t \leftarrow H_{t-1} \cup \{(\theta_t, A_t, O_t)\}$ {Update history}
- 35: Update prompt P_{t+1} with H_t
- 36: **if** stop_flag is TRUE **then**
- 37: **BREAK**
- 38: **end if**
- 39: **end for**
- 40: **return** S_{linked}
- 41: **Step 3: SQL Generation**
- 42: $SQL_{predicted} \leftarrow \pi_{SQL}(Q, S_{linked})$ {The entire SQL generation process operates **only on** the final simplified linked schema S_{linked} }

- **SQL Execution Error** For any other SQL execution error (e.g., *syntax error, non-existent column*), R_{SQL} contains the specific, verbatim error message generated by the SQL execution engine. This direct feedback is crucial for the agent to debug its queries and refine its understanding of the schema:

```
1 [ERROR: column "user_id" does not exist]
```

Schema Vector Store Environment (\mathcal{E}_{VS}) To facilitate semantic retrieval over schema elements, we construct a vector database indexing all columns in the schema. In practice, special care is taken to handle partitioned tables prevalent in BigQuery and similar data warehouses. The overall setup procedure consists of the following steps:

- **Handling of Partitioned Tables** In data warehouses like BigQuery, it is common to encounter partitioned tables (e.g., ‘table_20230101’, ‘table_20230102’, etc.) that share an identical underlying column structure but differ only in their names, typically reflecting a time-based partition. To maximize efficiency and avoid redundant embeddings for such tables, we implement a merging strategy. We first identify and group all tables that possess **exactly the same column schemas**. For a group of such partitioned tables $\{t_{p,1}, t_{p,2}, \dots\}$ that effectively represent the same logical entity with schema $\{c_1, \dots, c_n\}$, each unique column c_i from this shared schema is processed and **embedded only once**. This merging strategy significantly reduces the number of redundant entries in the vector store while preserving all necessary table context for retrieval. For example, in the **Spider 2.0-Lite** (Lei et al. 2025) benchmark, after applying this merging strategy, each database contains an **average of 559 columns, with a maximum of 6161 columns**. Crucially, without merging partitioned tables, the total number of columns could be as high as **100,000**.
- **Textual Representation for each Column** For each column $c_i \in S_{full}$ (after the merging process for partitioned tables), we construct a textual document by concatenating its core metadata. For each column, the constructed document includes: 1) **column name**; 2) **associated table names list**, i.e., a list containing the names of all tables that share this exact column schema (e.g., all table names for partitioned tables that were grouped); 3) **column data type**; 4) **column description** (if available) and 5) **declaration of primary and foreign keys**. An example column textual document is shown below:

```
1 Column: visitNumber;
2 Table: [ga_sessions_20170720,
3   ↪ ga_sessions_20170521, ...];
4 Type: INT64;
5 Description: The session number for this
6   ↪ user;
7 Primary Key: None;
8 Foreign Key: None;
```

- **Column Vector Store Indexing** Each column document (with all texts are lowercased) is embedded into a 1024-dimensional dense vector using the bge-large-en-v1.5 (Chen et al. 2024) model, a BERT-based (Devlin et al. 2019) open-source embedding model. These column embeddings are then stored in a vector database, specifically using Faiiss (Johnson, Douze, and Jégou 2019) for its Approximate Nearest Neighbor (ANN) (Liu et al. 2004) index. Each vector entry maintains a link to its comprehensive column metadata for downstream retrieval and formatting. To **quantify the construction overhead of this schema vector store environment per database**, we measure the time required for its construction on **one H100 GPU** using a **batch size of 1024**. For the **Spider 2.0-Lite** (Lei et al. 2025) benchmark, encompassing **158 databases**, the total time required for programmatic column document construction, embedding, and indexing into the vector database are **222.8 seconds**, averaging a mere **1.4 seconds per database**. Similarly, for the **BIRD** (Li et al. 2023c) benchmark, comprising **11 databases**, the complete environment setup takes **4.4 seconds**, averaging an even faster **0.4 seconds per database**. It underscores the rapid and low-cost deployment capabilities of our schema vector store.

- **Column Retrieval** Upon constructing the column vector store, a given natural language query is first encoded into a query vector using the same text encoder (i.e., bge-large-en-v1.5). This query vector is then utilized to retrieve the top- K most semantically similar columns from the vector store via Approximate Nearest Neighbor (ANN) search, ranked by **cosine similarity**, with a built-in mechanism to exclude any columns that have been previously returned. **On average, encoding the query and performing the retrieval takes only 0.05 seconds on one H800 GPU**. For these retrieved columns, we identify their corresponding tables, or all associated table names for partitioned tables, and form a relevant schema subset, which is formatted and presented in the M-Schema (Gao et al. 2025) style. **M-Schema** is a semi-structured textual representation for schema, explicitly identifies hierarchical relationships between databases, tables, and columns using specific tokens, and provides comprehensive details such as column names, data types, primary/foreign keys, detailed descriptions, and sampled values. Below is an example of M-Schema for a partitioned table:

```
1 [DB_ID] ga360
2 # Table ga_sessions_20160810
3 (visitNumber: INT64,
4   ↪ Examples:[2,1,1,1,1], The session
5   ↪ number for this user. If this is the
6   ↪ first session, then this is set to
7   ↪ 1)
4 (fullVisitorId: STRING,
5   ↪ Examples:["1906","7880","5836",
6   ↪ "6743","0244"], The unique visitor ID.)
6 (date: STRING,
7   ↪ Examples:["20160810","20160810",
8   ↪ "20160810","20160810","20160810"]): The
9   ↪ date of the session in YYYYMMDD
10  ↪ format.)
```

```

8   (visitStartTime: INT64,
9    → Examples: ["1470817379", "1470856696",
10   "1470873918", "1470816856", "1470883270"],
11   → The timestamp (expressed as POSIX
12   time).)

```

Details of Action Space

In the main paper, we introduce the core actions that empower AutoLink’s autonomous agent. This section provides a more comprehensive overview of three actions (i.e., `@explore_schema`, `@retrieve_schema` and `@add_schema`) within the agent’s action space. For each action, we offer concrete examples of its usage and detail its specific utility. Readers can also refer to AutoLink’s pseudocode in Algorithm 1 for a better understanding of these actions.

1. `@explore_schema` This action is designed to interact with the Database Environment (\mathcal{E}_{DB}) by executing exploratory SQL queries mainly against the full schema metadata S_{full} . Exploratory queries under this action can be categorized into two main aspects: exploring value distributions and exploring schema structure. For example, in terms of exploring value distributions, the agent can execute queries to verify specific values, sample data, or analyze value ranges, which helps confirm whether the values involved in the problem exist in the target columns:

```

1   -- Check sample values containing specific
2   → keywords (e.g., "INTOX")
3   SELECT DISTINCT descriptor
4   FROM incidents_2016
5   WHERE descriptor LIKE '%INTOX%'
6   LIMIT 5;
7
8   -- Sample values to understand data format
9   SELECT date, timestamp, time
10  FROM incidents_2016
11  LIMIT 5;
12
13  -- Analyze value range of a date column
14  SELECT MIN(date), MAX(date)
15  FROM incidents_2016;

```

In terms of exploring schema structure, the agent can check table structures, search for relevant tables or columns using metadata queries, which helps clarify the database’s organizational structure. Note that **INFORMATION_SCHEMA** is a standard metadata repository supported by most relational databases (e.g., BigQuery, Snowflake) for querying schema details, while **PRAGMA** is a database-specific command primarily used in SQLite for retrieving metadata like table structures. Specific exploration examples are as follows:

```

1   -- Explore table structure by sampling rows
2   → to observe column types and data
3   → patterns
4   SELECT * FROM inpatient_charges_2014

```

```

3   LIMIT 5;
4
5   -- Search for columns related to target
6   → semantics using INFORMATION_SCHEMA
7   SELECT column_name
8   FROM new_york.INFORMATION_SCHEMA.COLUMNS
9   WHERE column_name LIKE '%trips%';
10
11  -- Filter columns by dual conditions (table
12  → attributes + column semantics) via
13  → metadata
14  SELECT column_name
15  FROM census.INFORMATION_SCHEMA.COLUMNS
16  WHERE table_name LIKE '%tract%' AND
17  → table_name LIKE '%2018%'
18  AND LOWER(column_name) LIKE '%income%'
19  LIMIT 5;
20
21  -- Identify relevant reference tables by
22  → keyword matching in table names
23  FROM
24  → new_york_taxi.INFORMATION_SCHEMA.TABLES
25  WHERE LOWER(table_name) LIKE '%zone%' OR
26  → LOWER(table_name) LIKE '%borough%'
27  LIMIT 5;
28
29  -- check column details of a table using
30  → database-specific metadata commands
31  → (e.g., PRAGMA for SQLite)
32  PRAGMA table_info(results);

```

2. `@retrieve_schema` The action empowers the agent to actively search for missing schema elements using the Schema Vector Store Environment (\mathcal{E}_{SV}). Unlike simple query rewrite methods such as CHESS (Talaei et al. 2024), this action is dynamic: the agent formulates a new natural language query for the vector store by combining the original user question with contextual information, including all table names and partially linked schema elements. Crucially, this query can directly be an *inferred virtual column name*, *a description of a potentially missing column*, or an *abstract concept/phrase* that the agent generates to target the discovery of missing schema elements. This capability is vital because it transcends the limitations of simple query rewriting, which often struggles with ambiguous terms or when relevant schema elements are not directly discoverable from the literal words in the user’s question, enabling a more sophisticated and targeted search.

For example, consider a user asking, “*What’s the score?*” On its own, this question is ambiguous; “*score*” could refer to a *game score*, a *credit score*, a *performance score*, or a *test score*. A simple query rewriter would struggle to map “*score*” to a specific, actionable database column, as its meaning is entirely context-dependent. However, if the agent knows there is a table named **students**, the question’s intent immediately becomes clear: the user is asking about *student academic scores*. In this context, the semantic search can specifically target columns like *exam_score*, *quiz_score*, or *final_grade*, even if these specific columns are not explicitly present in the partial schema.

```

1 @retrieve_schema(`exam score, quiz score or
→ final grade`);

```

3. @verify_schema This action takes the currently linked schema elements S_{linked} as a working hypothesis and constructs a **minimal executable** SQL query to test whether this hypothesis is sufficient to answer the user’s question. The purpose of this action is not to obtain the final query result, but to convert the database engine’s execution outcome—success or error messages—into high-precision diagnostic signals. For example, *no such column*: X clearly indicates a missing column X ; *no such table*: T points to an un-linked table. Therefore, @verify_schema often forms a feedback loop with @retrieve_schema and @explore_schema: errors provide semantic clues for targeted retrieval, which are then incorporated into S_{linked} via @add_schema, followed by re-verification.

An example of verification is as follows, suppose the user asks, “*How many arrests occurred in 2016?*” The agent hypothesizes that the relevant table is *incidents_2016* and that the arrest indicator column might be *is_arrest*. It then issues the following verification query:

```

1 -- Verification query (minimal hypothesis)
2 SELECT COUNT(*)
3 FROM incidents_2016
4 WHERE is_arrest = 1;

```

If the execution returns *Error: no such column: is_arrest*, the agent treats this as a precise signal and invokes @retrieve_schema (e.g., *searching for a “column indicating arrest status”*). After adding the retrieved column via @add_schema, it re-runs a simplified verification query to confirm that the schema has been sufficiently corrected.

4. @add_schema This is the agent’s mechanism for committing discoveries. After actions with feedback yields new, relevant schema elements, the agent can adopt this action to add them to its final candidate set. We encourage agent to use this action frequently to add sufficient schema elements to improve schema linking recall, using the format of *table.name.column_name*, as shown below.

```

1 @add_schema(`orders.id; orders.prod_sku;
→ users.is_active; products.cat_id`);

```

Detail of SQL Generation

Given the final linked schema S_{linked} produced by our agent-based retrieval and linking process, we formalize the subsequent SQL generation step as a conditional sequence generation problem. The overall pipeline consists of multiple modules designed to enhance both syntactic and semantic correctness:

SQL Candidate Generation via LLM Policy Let π_{SQL} denote a large language model (LLM) policy. Given the user

question Q and the contextualized schema S_{linked} , we generate a set of N SQL candidates (5 in this paper):

$$\{\text{SQL}_i\}_{i=1}^N = \pi_{\text{SQL}}(Q, S_{\text{linked}}) \quad (6)$$

where N is the number of self-consistency samples (Wang et al. 2023). Each candidate is independently generated by sampling from the LLM in stochastic decoding mode (temperature sampling).

Iterative Syntactic Correction Each sampled SQL candidate SQL_i is refined through an iterative syntactic correction process, also facilitated by the LLM policy π_{SQL} in a multi-turn dialogue. We denote by t the maximum number of dialogue turns. Let \mathcal{C}_0 be the initial conversation context, which includes the user query Q , the linked schema S_{linked} , and the initial SQL candidate $\text{SQL}_i^{(0)} = \text{SQL}_i$. At each turn j , we update the conversation context \mathcal{C}_j by incorporating the previously proposed SQL statement and its execution error:

$$\mathcal{C}_j = \mathcal{C}_{j-1} \cup \{(\text{SQL}_i^{(j-1)}, \text{error}_j)\}. \quad (7)$$

The LLM then produces the revised SQL statement:

$$\text{SQL}_i^{(j)} = \pi_{\text{SQL}}(\mathcal{C}_j). \quad (8)$$

This process continues until $\text{SQL}_i^{(j)}$ successfully executes without errors or until the maximum number of dialogue turns t is reached. The final corrected version after k turns is denoted by:

$$\text{SQL}_i^{\text{corr}} = \text{SQL}_i^{(k)}, \quad (9)$$

where $k \leq t$ represents the iteration at which the statement is deemed correct or the dialogue terminates.

Majority Voting via Execution-Based Output Grouping To further enhance robustness, we employ a majority voting strategy grounded in the execution results of SQL candidates. The complete procedure is as follows:

- **Step 1: Execution-Based Grouping.** Consider the set of syntactically valid SQL candidates $\mathcal{F} = \{\text{SQL}_i^{\text{corr}}\}_{i=1}^N$. For each candidate, we execute it on the database and group all candidates that yield identical outputs (i.e., same result set). Formally, let \mathcal{G}_j denote the set of candidates corresponding to the j -th unique query output.
- **Step 2: Group Selection by Majority.** Identify the group(s) with the largest cardinality. Let \mathcal{G}_{\max} denote the set of group(s) attaining the maximal size:

$$\mathcal{G}_{\max} = \left\{ \mathcal{G}_j \mid |\mathcal{G}_j| = \max_k |\mathcal{G}_k| \right\} \quad (10)$$

- **Step 3: Final SQL Selection.**

- *Single Majority Group*: If $|\mathcal{G}_{\max}| = 1$ (i.e., only one group has the majority count), we randomly select one SQL from this group as the final output.
- *Multiple Majority Groups (Tie)*: If multiple groups share the maximal size, we select one representative SQL candidate from each tied group and aggregate these representatives into the set \mathcal{S}_{tie} . For every unordered pair $(\text{SQL}_a, \text{SQL}_b)$ in \mathcal{S}_{tie} , we prompt an LLM

(provided with the user question, schema, candidate SQL, and their execution results) to select the better SQL between them. Each win counts as one point. The SQL with the highest total number of pairwise wins is selected. In the rare event of a tie in pairwise wins, we randomly select one among the top scorers.

Formally, the final selection can be expressed as:

$$\text{SQL}^* = \begin{cases} \text{RandomSelect}(\mathcal{G}_{\max}), & \text{if } |\mathcal{G}_{\max}| = 1 \\ \underset{\text{SQL} \in \mathcal{S}_{\text{tie}}}{\text{argmax}} \text{PairwiseWins}(\text{SQL}), & \text{otherwise} \end{cases} \quad (11)$$

where $\text{PairwiseWins}(\text{SQL})$ denotes the number of times a SQL candidate is preferred over others in LLM-based pairwise comparisons, and \mathcal{S}_{tie} is the set of all SQLs in the tied majority groups.

More Experiment Details

Experimental Setup

In the main experimental results of schema linking, we set $\text{top-}n$ to 100 for initial schema retrieval in Spider 2.0-lite, while for the smaller Bird dataset, we set $\text{top-}n$ to 30. For the `@retrieve_schema` action, $\text{top-}m$ is fixed at 3, and the maximum interaction turn limit is set to 10. Additionally, for the DE-SL and CE-SL methods, we set retrieval $\text{top-}k$ to 200 on Spider 2.0-lite and 40 on the Bird dataset. The number of sampling decoding is set to 5.

In the SQL generation phase, we adopt a self-consistency sampling approach with a temperature setting of 1.0, generating 5 SQL candidates. Subsequently, these candidates undergo iterative syntactic correction, with a maximum of 5 dialogue turns allowed for each candidate to integrate execution feedback and refine the SQL until it successfully executes. On the Bird dataset, we utilize DeepSeek-V3 for generating SQL. However, on the Spider 2.0-Lite dataset, which demands more complex logical reasoning, we employ DeepSeek-R1.

Benchmark

- **Bird** (Li et al. 2023c) is a cross-domain dataset designed to evaluate the impact of extensive database contents on text-to-SQL parsing. It comprises over 12,751 unique SQL question pairs, spans 95 large databases, and has a total size of 33.4 GB. The dataset encompasses more than 37 specialized domains. All our experiments on BIRD are conducted on the Bird Dev dataset.
- **Spider 2.0-Lite** (Lei et al. 2025) serves as a benchmark to assess the performance of language models on complex enterprise-level text-to-SQL tasks. As an upgrade from Spider 1.0 (Yu et al. 2018), it focuses on more intricate SQL generation tasks across various databases and SQL dialects. Spider 2 includes multiple versions—Spider 2.0, Spider 2.0-Lite, and Spider 2.0-Snow—tailored for different database systems such as BigQuery, Snowflake, and SQLite. Given the inherent complexity of Spider 2.0-Lite, which supports these three distinct database dialects, all our experiments concerning

Spider 2.0 were exclusively conducted on the Spider 2.0-Lite dataset. Its main features include a complex environment with over 3,000 columns, multi-step SQL generation requiring handling long contexts and complex reasoning, and a notably challenging nature, where even advanced models like GPT-4 achieve only 6.0% accuracy, significantly lower than the 86.6% success rate of Spider 1.0. Our schema linking evaluations are performed on a subset of Spider 2.0-Lite. This subset, comprising 250 examples, is crucial as it provides the necessary ground truth SQL for evaluating schema linking recall.

Baselines

- **DE-SL:** DE-SL employs a dual-encoder architecture to encode the question and schema elements separately. The relevance between questions and schema components is computed based on the similarity of their encoded representations.
- **CE-SL:** CE-SL uses a cross-encoder model that jointly encodes pairs of question tokens and schema elements. This allows the model to directly model intricate interactions between the question and schema components, generally leading to higher linking accuracy at the cost of increased computation.
- **MCS-SQL:** MCS-SQL (Lee et al. 2025) expands the schema linking search space by utilizing multiple prompts and leveraging the sensitivity of large language models (LLMs) to in-context learning (ICL) exemplars. By decoding the LLM multiple times with diverse prompts, MCS-SQL obtains a broader range of candidate schema elements and ultimately selects the most relevant ones for the given question.
- **SQL-To-Schema:** SQL-to-Schema (Yang et al. 2024) generates an initial SQL query by leveraging the complete database schema. It then extracts the involved tables and columns from the generated SQL to construct a concise schema tailored to the input question.
- **RSL-SQL:** RSL-SQL (Cao et al. 2024) integrates several techniques such as bidirectional schema linking, contextual information enhancement, a binary selection strategy, and multi-turn self-correction. These approaches collectively enable more robust schema linking, leading to improved performance on text-to-SQL tasks.
- **LinkAlign:** LinkAlign (Wang, Liu, and Yang 2025) is a framework designed to systematically address schema linking challenges and effectively adapt existing baseline models to real-world environments. The framework comprises three key stages in the form of multi-agent discussion: multi-turn semantic-enhanced retrieval, irrelevant information isolation, and schema extraction enhancement.
- **CHESS:** Context-aware, Hierarchical and Extensible SQL Synthesizer is an end-to-end text-to-SQL system designed for real-world and complex databases (Talaei et al. 2024). It proposes an efficient process centered on a large language model and divided into three major modules: entity and context retrieval, schema selection, and

SQL generation. This process can fully utilize database context information and enhance the accuracy and practicality of text-to-SQL in large-scale and heterogeneous schemas.

- **Spider-Agent:** Spider-Agent is a tool-call-based baseline model introduced in Spider 2.0 (Lei et al. 2025). It serves as a crucial benchmark for text-to-SQL tasks on complex databases, facilitating the evaluation of different large language models (LLMs) under unified settings.
- **ReFoRCE:** ReFoRCE (Deng et al. 2025) is a leading Text-to-SQL agent on the Spider 2.0 benchmark. It addresses challenges in complex, real-world databases through schema compression, self-refinement, consensus voting, and execution-guided exploration, achieving state-of-the-art results across multiple SQL dialects.
- **MAC-SQL:** MAC-SQL (Wang et al. 2025) is a multi-agent Text-to-SQL framework, featuring a Selector for schema selection, a Decomposer for stepwise SQL generation, and a Refiner for query correction based on execution results.
- **TA-SQL:** TA-SQL (Qu et al. 2024) is a text-to-SQL framework that mitigates schema- and logic-based hallucinations via a Task Alignment strategy, including modules for task-aligned schema linking and logical synthesis, and is evaluated with GPT-4 on the BIRD dataset.

More Experiment Results

Method	SRR	\bar{C}	Avg. Tokens
MCS-SQL _{N=1}	46.0	34.52	33.80K
MCS-SQL _{N=2}	54.4	39.04	67.50K
MCS-SQL _{N=3}	56.0	42.86	101.50K
MCS-SQL _{N=4}	57.6	44.10	135.10K
MCS-SQL _{N=5}	58.8	45.15	168.90K
SQL-to-Schema _{N=1}	45.2	32.70	34.68K
SQL-to-Schema _{N=2}	53.6	40.31	69.18K
SQL-to-Schema _{N=3}	58.4	35.69	103.50K
SQL-to-Schema _{N=4}	62.4	47.44	137.76K
SQL-to-Schema _{N=5}	64.0	49.03	171.90K

Table 5: Multi-turn results of MCS-SQL and SQL-to-Schema on Spider 2.0-Lite. N represents the times of iterations for decoding.

Limitations of Increasing Decoding Time in Iterative Schema Linking

Table 5 reports the strict recall rate, the average number of recalled columns, and the average token consumption for both **MCS-SQL** and **SQL-to-Schema** methods as the number of sampling-decoding turns increases on Spider 2.0-Lite. From the results, we observe a clear saturation effect in both strict recall and the number of columns: At the beginning, increasing the number of N from 1 to 2 or 3 brings a substantial gain in both recall and average recalled columns. However, **as the number of iterations continues to grow,**

the improvement becomes marginal—both metrics gradually approach an upper bound. Specifically, for MCS-SQL, moving from 1 time to 5 times increases SRR from 46.0% to 58.8%, and average columns from 34.52 to 45.15; for SQL-to-Schema, SRR increases from 45.2% to 64.0%, and average columns from 32.70 to 49.03. Nevertheless, the incremental improvement in the last several rounds is very limited.

In contrast, the average token consumption grows almost linearly with the number of N , as more candidates are sampled, decoded, and processed with each additional iteration. This suggests that, although both methods allow one to control the iteration count (N), **it is difficult to directly control the scale or to reach a recall-quantity comparable to our approach without incurring significantly higher token cost.**

Overall, these findings indicate a practical limitation of simply increasing the number of decoding rounds in such iterative methods: recall and result size eventually saturate, while resource (token) consumption continues to increase, making it challenging to efficiently match the effectiveness and scalability of our proposed approach.

Analysis of Hyperparameter

In this section, we continue to analyze the influence of the hyperparameters **Max Turn** (The maximum number of dialogue iteration rounds of the agent) and **top- m** in `@retrieve_schema`.

Max Turn	Avg. Turn	SRR	Avg. Tokens	\bar{C}
4	3.72	89.2	20.1K	138.4
6	4.65	90.2	20.7K	152.0
8	5.01	91.0	20.8K	153.5
10	5.79	91.2	21.2K	159.6

Table 6: Impact of Max Turn on SRR, Token Cost and Retrieved Columns.

Impact of Max Turn. Table 6 evaluates the effect of varying the maximum number of agent dialogue turns (Max Turn) in **AutoLink** on SRR, token cost, and average recalled columns. We observe that even with a relatively small Max Turn (e.g., 4 or 6), the method already achieves strong recall (SRR 89.2 and 90.2 respectively) and covers the majority of relevant columns. This demonstrates that **the agent is able to efficiently explore the schema and discover important elements within the first few interaction rounds.**

As Max Turn increases, both SRR and average recalled columns exhibit only incremental gains—SRR increases by just 2 (from 89.2 to 91.2) and columns by about 21 (from 138.4 to 159.6) as Max Turn moves from 4 to 10, indicating a clear saturation effect. Importantly, the average number of turns taken by the agent grows much more slowly than the allowed maximum (from 3.72 to 5.79 as Max Turn rises from 4 to 10), suggesting that in practice the exploration process tends to converge early, with later rounds contributing diminishing additional gains.

Additionally, the average token cost remains nearly unchanged as Max Turn increases, further confirming that most schema expansion and information acquisition occur in the initial rounds. Overall, these findings indicate that **AutoLink is capable of rapidly performing effective schema exploration, and strict limits on Max Turn are not necessary to achieve near-optimal recall and coverage.**

top- m	SRR	Avg. Tokens	\bar{C}
1	89.2	20.7K	152.28
2	89.6	21.0K	156.06
3	91.2	21.2K	159.40

Table 7: Impact of top- m in retrieval on SRR, token cost and retrieved columns.

Impact of Top- m in Retrieval. Table 7 examines the impact of the retrieval column number (top- m) per `@retrieve_schema` action on SRR, token cost, and column coverage. Notably, even with a small top- m setting (e.g., $m=1$ or 2), our method already achieves a high recall rate (SRR of 89.2% and 89.6%, respectively) and a substantial number of recalled columns, with little difference compared to larger m . As m increases from 1 to 3, SRR improves only slightly (to 91.2%), accompanied by marginal increases in both average tokens and column counts. This result demonstrates that **increasing top- m does not lead to significant growth in computational cost or excessive schema expansion.**

The relatively stable performance across different m values indicates that the agent is able to issue precise and targeted semantic queries in `@retrieve_schema` steps, retrieving most relevant columns with a small number of candidates per action. Therefore, **it is not necessary to set a large m value to achieve strong recall, as the agent’s high-quality search queries already ensure efficient coverage of the required schema elements.** This highlights the effectiveness and efficiency of our agent-based schema retrieval framework.

Analysis of Trade-off Comparison

As shown in Figure 4, we compare **AutoLink** (initial schema top- n set to 5, 10, 20, 50, 100), **MCS-SQL** and **SQL-to-Schema** (the times of iterations for decoding N set to 1 and 5), **BGE-Large** and **BGE-reranker** (retrieval top- k set to 50, 100, 200). Our method AutoLink demonstrates a clear advantage in strict recall rate across varying scales of recalled columns compared to baseline approaches. Specifically, when compared with MCS-SQL and SQL-to-Schema, our method achieves substantially higher strict recall rates. The recall columns returned by MCS-SQL and SQL-to-Schema remain relatively limited, which is primarily constrained by the design of these methods. Both baselines rely on repeatedly decoding with LLMs to extract potentially relevant schema elements. While increasing the number of decoding iterations from 1 to 5 improves both the number of recalled columns and the strict recall rate, the improvement plateaus quickly and is accompanied by a dramatic

increase in token consumption. Furthermore, **these LLM-based methods lack fine-grained control over the recall column size, making a fair comparison with AutoLink under the same recall scale infeasible.**

In contrast, when compared with retrieval-based methods such as BGE-Large and BGE-Reranker, AutoLink consistently achieves higher strict recall rates under the same average number of recalled columns. This highlights the effectiveness of our approach in providing both scalable recall and high precision.

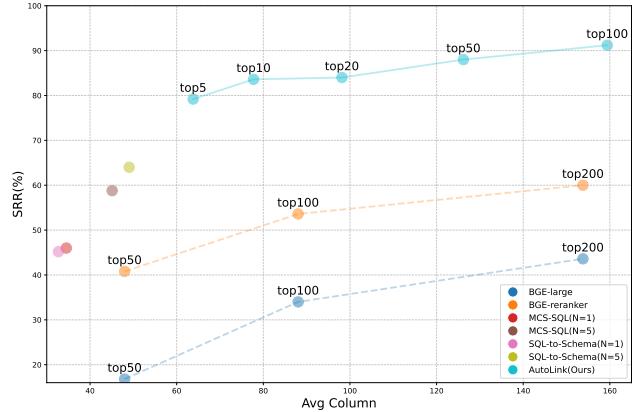


Figure 4: Trade-off comparison between recall columns per instance and strict recall rate across different methods.

Results on Leaderboard of Spider 2.0-Lite

As shown in Figure 5, we present the performance of our method, **AutoLink**, on the Spider 2.0-Lite leaderboard. At the time of submission, our approach achieves the second-best EX overall. Notably, when compared with other methods utilizing the same DeepSeek-R1 model, AutoLink attains **state-of-the-art** performance.

SQL Generation Ablation

Method	EX	Δ EX
Base Generation	23.47	–
+ Iterative Correction	31.34	+7.87
+ Majority Voting	34.97	+3.63

Table 8: Ablation for SQL generation on Spider 2.0-Lite.

Table 8 presents the ablation results for SQL generation on the Spider 2.0-Lite dataset. Starting from the base generation model, which achieves an EX of 23.47, we observe a substantial improvement when iterative correction is applied, raising the EX score by 7.87 points to 31.34. This demonstrates the effectiveness of iterative correction in refining initial predictions and mitigating errors. On Spider 2.0-Lite, due to the complexity of the problem, we found that the model was prone to low-level syntax errors during the process of generating SQL. Therefore, SQL corrections

Rank	Method	Score
1 May 22, 2025	ReFoRCE + o3 <i>Hao AI Lab x Snowflake</i> [Deng et al. '25]	37.84
2 Jul 19, 2025	AutoLink + Deepseek-R1 <i>Anonymous</i>	34.92
3 Jul 10, 2025	RSL-SQL + o3 <i>HUST VLR Lab</i> [Cao et al. '24]	33.09
4 Apr 27, 2025	LinkAlign + DeepSeek-R1 [Wang et al. '25]	33.09
5 Jul 10, 2025	RSL-SQL + DeepSeek-R1 <i>HUST VLR Lab</i> [Cao et al. '24]	30.53
6 Jan 28, 2025	ReFoRCE + o1-preview <i>Hao AI Lab x Snowflake</i> [Deng et al. '25]	30.35
7 Mar 16, 2025	Spider-Agent + Claude-3.7-Sonnet-20250219-Thinking	28.52
8 Mar 31, 2025	Spider-Agent + Claude-4-Sonnet-20250514	27.79
9 Jul 10, 2025	RSL-SQL + DeepSeek-V3 <i>HUST VLR Lab</i> [Cao et al. '24]	26.14
10 Mar 8, 2025	Spider-Agent + Claude-3.7-Sonnet-20250219	25.41
11 Mar 28, 2025	LinkAlign + DeepSeek-V3 [Wang et al. '25]	24.86

Figure 5: The results on leaderboard of Spider 2.0-Lite. Among all the submitted methods, AutoLink ranked second.

based on the execution results are necessary. Further incorporating majority voting leads to an additional boost, increasing the EX to 34.97, a gain of 3.63 points over the previous step. These results highlight the complementary benefits of both iterative correction and majority voting, showing that the combination of these strategies significantly enhances the robustness and overall performance of the SQL generation process.

Prompt Templates

In this section, we present the detailed prompt templates used for the various key modules in our system, namely: (1) Schema Linking in Figure 6, (2) SQL Generation in Figure 7, (3) Syntactic Correction in Figure 8, and (4) SQL Selection in Figure 9. Each template is designed to guide the Large Language Model (LLM) effectively for its specific subtask.

AutoLink Agent Prompt

You are an intelligent assistant designed to help identify all relevant schema elements (tables and columns) in a large, unfamiliar database to answer a user's natural language question. You do **not** have access to the full database schema. Instead, you can interact with two external environments:

1. **Database Environment ('DB Environment')**: - This lets you run SQL queries to explore schema metadata, table/column names, table relationships, keys, or sample values.
2. **Schema Vector Store Environment ('VS Environment')**: - This lets you semantically search for the most relevant columns using a natural language query. You get back structured schema snippets (column, parent table, type, optional description, sample values).

Your goal is to maximize recall of truly relevant schema elements for the question, using as few, but as effective, interactions as possible.

Interaction rules:

- Work in multiple turns (each with reasoning and actions).
- At every turn, you will receive:
 - The user's question.
 - The complete list of all table names in the database.
 - The current linked schema (already discovered schema elements, with full metadata).
 - The complete history of your reasoning, actions, and all observations so far.
 - The latest feedback (result or error) from the environment.
- In each turn:
 - Carefully read the full context
 - Write your reasoning in '`<think></think>`' tags: What schema or information is still missing? What do you want to find or verify?
 - Output one or more actions in '`<actions></actions>`' tags;

Available actions:

- `@explore_schema(sql_query)` Use SQL to explore tables, columns, structure, or preview data in the DB Environment. Examples: list columns in a table, show sample values, find tables/columns matching certain keywords, query table or column descriptions, check foreign/primary keys, etc.
- `@retrieve_schema(natural_language_query)` Use the VS Environment to semantically search for up to {top-K} columns by describing the desired concept, column, or property in plain language.
- `@verify_schema(sql_query)` Attempt a full SQL query to answer the user's question using ONLY the currently linked schema. Use the result or errors to identify what's still incomplete or missing.
- `@add_schema(table.column; ...)` After discovering new, relevant columns (using retrieve or explore), add them to the linked schema. Always use full `table.column` names, separated by semicolons. Cannot be used alone in a turn—pair with another action.
- `@stop` Use when the linked schema comprehensively covers everything needed to answer the question, or if {Max_Turn} turns have been reached.

How to behave:

- In early turns, focus on obvious gaps in the linked schema by using semantic retrieval or basic table/column exploration.
- If you suspect that critical columns or concepts are missing, use `@retrieve_schema` with the relevant keyword or phrase.
- Use `@explore_schema` for table or structural exploration, such as unknown table layouts or to clarify join conditions or key relationships.
- Use `@verify_schema` to confirm whether the currently linked schema suffices or to uncover missing schema via error feedback.
- Whenever you get new candidate columns that are likely relevant, add them to your linked schema using `@add_schema` (paired with a feedback action in the same turn).
- End with `@stop` only if you are confident no more relevant schema elements are missing.

Always output in the following format (every turn):

`<think> [Your detailed reasoning about what is currently known, what is uncertain or missing, and what actions you plan to take and why. Be concise and explicit.] </think>`
`<actions> @ [your actions, separated by semicolons, with required arguments] </actions>`

REMEMBER:

- Justify your actions in '`<think>`'.
- Never use `@add_schema` by itself in a turn—always pair with a feedback-providing action or `@stop_action`.
- Read feedback (including error messages) to refine your next step.
- Use semantically meaningful, natural language queries in `@retrieve_schema`—don't just repeat the user's question if you can guess a more relevant phrase.
{Initial Input}

You are now ready to begin. Each time, read the history, the schema so far, and generate a new '`<think>`' and '`<actions>`' block in the required format.

Figure 6: Template of agentic schema linking process.

SQL Generation Prompt

You are a professional data engineer skilled in translating complex natural language questions into accurate and efficient SQL queries. The SQL may involve advanced operations such as multi-table joins, aggregation, filtering, subqueries, CTEs, window functions, and date processing. You must generate SQL in {SQL_TYPE} dialect.

Question:

{QUESTION}

Database Schema and External Knowledge:

{Database Schema}

🌐 Step-by-Step Reasoning

Step 1: Deeply Understand the Question Intent

1. Clearly summarize the core objective of the question.
2. Decompose the question into well-defined sub-problems.
3. Explicitly list out all operations required: aggregation, filtering, sorting, joins, date manipulations, ranking, window functions, etc.

Step 2: Identify Relevant Tables and Columns

1. Precisely identify relevant tables and columns required to answer the question based on clear evidence.

2. Clearly specify any explicit constraints from the question (dates, numerical thresholds, text patterns).

3. Highlight any implicit constraints or potential ambiguities that need verification.

Step 3: Design the SQL Query Structure

Clearly outline the planned SQL structure:

* Specify if CTEs (WITH clause) are required. Follow syntax rigorously ('table_name AS (SELECT ...)').

* Clearly define SELECT, FROM, JOIN conditions, WHERE filters, GROUP BY/HAVING conditions, ORDER BY/LIMIT operations.

* Specify exact operations (UNNEST, ST_DISTANCE, window functions, etc.) needed.

Step 4: Logical Validation (Critical)

* Before generating the final SQL, explicitly verify that your designed SQL fully meets every constraint (explicit and implicit) mentioned in the original question.

* Clearly explain why your SQL logic is correct and how it satisfies the user's intent comprehensively.

Step 5: Write the Final SQL Query

* Ensure accurate parentheses pairing and commas placement.

* Annotate your SQL clearly using comments to explain each part.

⚙️ Apply Optimization Strategies

When writing the SQL query, consider the following optimization strategies:

{SQL_DIALECT_OPTIMIZATION}

- Execution result content:

- When asked something without stating name or id, return both of them. e.g. Which products ...? The answer should include product_name and product_id.\n"

- Make sure that the query content of the sql definitely includes what needs to be involved in the question, the execution result can be more than what is required by the question, but it must not be less.

🖨️ Output Format

In addition to outputting other information, you also need to return the generated SQL query in the following format:

```sql

Your SQL query

```

Make sure that all the SQLs is contained within ```sql``` and the last ```SQL``` contains the final complete SQL in your output.

Figure 7: Template of SQL generation.

SQL Correction Prompt

You are a professional data engineer skilled in translating complex natural language questions into accurate and efficient SQL queries. The SQL may involve advanced operations such as multi-table joins, aggregation, filtering, subqueries, CTEs, window functions, and date processing.

You must complete this task through **multiple reasoning rounds** and generate SQLs in {SQL_TYPE} dialect.

Database Schema and External Knowledge:

{PROMPT}

Question:

{QUESTION}

SQL Query:

{SQL}

✖ The SQL you generated encountered an error during execution.

Error Message:

{ERROR_MESSAGE}

Please help analyze the SQL and identify the root cause of the failure by following this structured checklist:

🔍 [1] Error Type Detection

- Based on the error message, determine the type of issue:
 - Syntax error (e.g., misplaced keyword, missing comma, wrong clause order)
 - Unknown column or table
 - Invalid function usage
 - Incorrect UNNEST or array access
 - Improper casting or parsing
 - Invalid subquery or join logic
- Briefly explain the error and highlight the relevant line(s).

📝 [2] Clause-by-Clause Syntax Review

Please examine each clause of the SQL query for syntax correctness:

SELECT Clause:

- Are all fields valid?
- Are nested fields accessed correctly (e.g., col.key, value.int_value)?
- Are aliases and expressions properly defined?

FROM Clause:

- Is the table name correct?
- If wildcard tables are used, is _TABLE_SUFFIX handled?
- Are commas or joins misplaced?

WHERE Clause:

- Are boolean conditions well-formed?
- Is the logic clear (no dangling AND/OR)?
- Are fields used here actually defined in the schema?
- JOINs or UNNESTs (if any):
 - Are all array fields unnested before access?
 - Are join conditions properly specified?

GROUP BY / HAVING / ORDER BY:

- Are aggregation fields valid?
- Does SELECT contain only grouped or aggregated expressions?

🔧 [3] Fix or Rewrite Suggestion

Based on your analysis above, propose a corrected version of the SQL query.

Or, describe how the query can be restructured to fix the issue.

💡 [4] Error Examples

- The error message include 'Cannot access field on ARRAY<STRUCT<...>>': check whether 'UNNEST' is missing or improperly used.
- 'Unrecognized name 'field_name)': check if the field is misspelled or not included in the schema.
- 'Invalid function <...>': check if the function is supported in the SQL dialect.
- 'Syntax error: Unexpected keyword': check SQL spelling, comma, and keyword position issues

⚙️ Apply Optimization Strategies

When writing the SQL query, consider the following optimization strategies:

{SQL_DIALECT_OPTIMIZATION}

Output Format:

```sql

Your fixed SQL query

```

Make sure that all the sqls is contained within ```sql``` and the last ```sql``` contains the final SQL in your output.

Figure 8: Template of iterative SQL syntactic correction.

SQL Selection Prompt

{Dialect} SQL tables, with their properties:
{Database_Schema}
Answer the question by {Dialect} SQL query only and with no explanation.
Question: {Question}
Two SQLs, the results of execution will be given.

Note:
It is unreasonable if all rows are null.
Select the best SQL query to answer the question correctly from the given two SQLs:
SQL1:
{sql1}
Execution result of the SQL1 (First 1000 rows limit 10,000 characters):
{re1}
SQL2:
{sql2}
Execution result of the SQL2 (First 1000 rows limit 10,000 characters):
{re2}

Output format:
Just output tag "SQL1" OR "SQL2", don't contain any external explanation.

Figure 9: Template of SQL selection.