LIST: Linearly Incremental SQL Translator for Single-Hop Reasoning, Generation and Verification

Kaiyuan Guan[1]*, Ruoxin Li[12]*†, Xudong Guo[13]†, Zhenning Huang[1], Xudong Weng[1], Hehuan Liu[1], Zheng Wei[1], Zang Li[1]
[1]Platform and Content Group, Tencent
[2]School of Data Science, Fudan University
[3]Department of Automation, Tsinghua University
{wilfredguan, ruoxinli, brucexdguo, zenithhuang, steveweng, mermaidliu, hemingwei, gavinzli}@tencent.com

Abstract

SQL languages often feature nested structures that require robust interaction with databases. Aside from the well-validated schema linking methods on PLMs and LLMs, we introduce the Linearly Incremental SQL Translator (LIST), a novel algorithmic toolkit designed to leverage the notable reasoning and tool interaction capabilities inherent in LLMs. LIST transforms complex SQL queries into grammatically verifiable sub-queries which are arranged sequentially to reflect single-hop reasoning steps, enhancing both the granularity and accuracy of database interactions. With in-context learning, our experiments demonstrated significant improvements, achieving notable performance of 60.56% and 56.32% on the BIRD dataset with GPT-4o and Llama-3-70B-Instruct. To the best of our knowledge, this achieves SOTA performance among non-schema linking methods, also surpassing a series of schema linking based approaches at a comparable or better cost.

1 Introduction

The rapid development of large language models has enabled strong generalization and reasoning capabilities (OpenAI, 2024; Anthropic, 2024), making it possible to tackle challenging tasks such as production-level text-to-SQL. This task, crucial for making database interactions accessible to non-technical users, has seen significant progress on benchmarks like SPIDER (Yu et al., 2019) and BIRD (Li et al., 2023).

We consider SQL generation to be a multi-hop reasoning problem, where models should progressively explore database elements, including tables, columns, and values, based on natural language input, while uncovering explicit or implicit relations (Popescu et al., 2003, 2004).

However, most existing methods modularize the schema linking process, treating the reasoning and generation ontological (Hong et al., 2024). For example, some approaches build connections between natural language and SQL using graph models or context-aware training techniques (Yu et al., 2018; He et al., 2019; Lei et al., 2020; Cao et al., 2021; Zhong et al., 2021; Cai et al., 2022), or employ intermediate representations to mitigate the impact of SQL's nested structures (Guo et al., 2019; Gan et al., 2021, 2022; Eyal et al., 2023). In addition, building schema linking modules with vector search or prompts for LLMs is another effective paradigm (Pourreza and Rafiei, 2023; Talaei et al., 2024; Pourreza et al., 2024). While schema linking modules with high accuracy and recall can significantly reduce costs and simplify challenges, the reasoning and analytical capabilities inherent to LLMs, which have the potential to tackle more challenging problems, remain largely underexplored.

Therefore, we intentionally avoid using schema linking modules, despite their effectiveness, and instead frame the task as explicit or implicit reasoning problems as shown and introduced in Figure 1. In conclusion, we pivot on:

(1) LIST (Linearly Incremental SQL Translator), an algorithmic toolkit, can parse Abstract Syntax Trees (AST) and generate verifiable sub-queries sequenced by single-hop reasoning steps. While we do not imply that LLMs reason in a strictly linear structure, they have shown stronger generalization abilities in single-hop reasoning (Yang et al., 2024).

(2) Through in-context learning, LIST guides the LLM to solve the question in a single-hop manner, showing results significantly superior to the baseline without any forms of training. Our work leverages the notable reasoning capabilities with intermediate steps, similar to chain-of-thought (CoT) (Wei et al., 2023) and tree-of-thought (ToT) (Yao et al., 2023), while providing structured guidance and verification for each step. At the same time, LIST provides the potential for dynamic exploration, generation, and verification.

*Equal contribution.
†Work done during the internship at Tencent.

Findings of the Association for Computational Linguistics: ACL 2025, pages 24886–24897

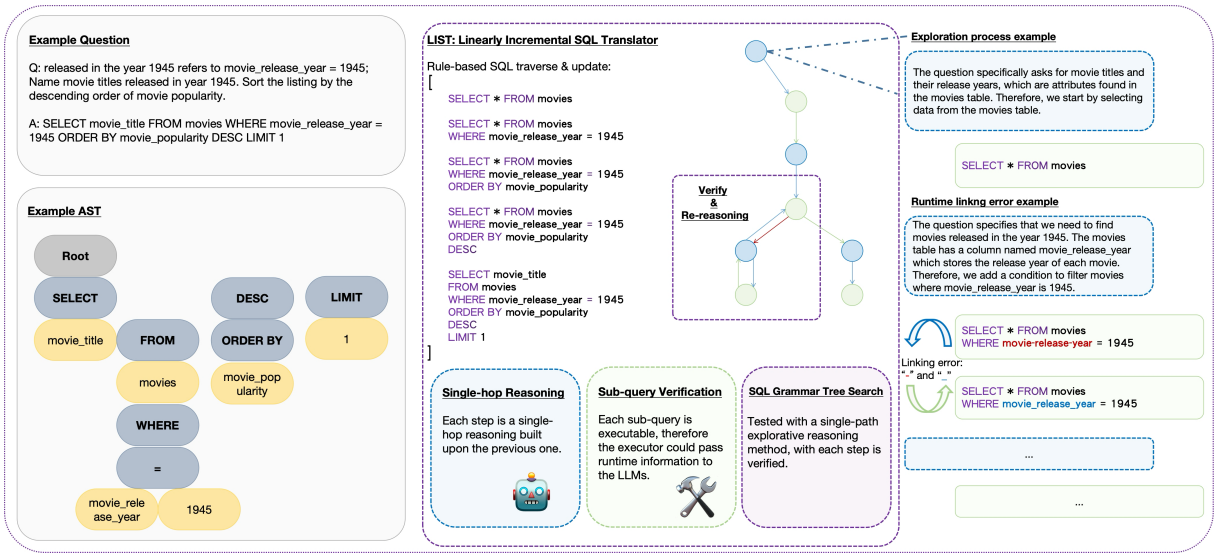

Figure 1: Overview of LIST and its implementation with in-context learning. **Example Question** is the first sample in BIRD-train set. **Example AST** represents the abstract syntax tree of an example SQL query, where the root node is shown in gray, operators are depicted in blue, and database elements are highlighted in yellow. The **LIST** section in the middle illustrates how SQL queries are decomposed into single-hop reasoning processes, which consist of minimal verifiable incremental steps formed by a set of keyword(s), operator(s), and database element(s). LIST is utilized to construct few-shot examples for dividing reasoning into sub-problems, guiding single-hop generation, and decompose outputs for verification.

2 Linearly Incremental SQL Translator

LIST starts from the SQL syntax tree, which is parsed and scheduled by the engine into an execution order. We will progressively provide: (1) how LIST produces verifiable single-hop reasoning steps (2) implementations of LIST with an in-context learning manner.

2.1 Formation of Single-Hop Reasoning

Given a natural language query Q and database structure information DB (including mapping relationships, columns, tables, etc.), the goal is to input Q and DB into a large language model (LLM). The LLM will reason to derive the corresponding SQL query Y.

Let the input $I = \{Q, DB\}$, then the LLM will output a series of steps S, which can be represented as:

$$S = \{S_1, S_2, \ldots, S_n\} = R(I) \quad (1)$$

Define $o = \{\text{select, join, from, } \ldots\}$, which includes database operations such as selection, joining, and specifying data sources. Define $e = \{\text{cols, cons, tables, } \ldots\}$ which e represents various database elements such as columns (cols), constraints (cons), and tables. Then,

$$\text{SQL} = f_{o_n}(f_{o_{n-1}}(\ldots f_{o_1}(e_1), e_{n-1}), e_n) \quad (2)$$

Now, we can represent the nested structure of the function $f(\cdot)$ in the following more concise manner:

$$\begin{aligned} F_1(E_1) &= f_{o_1}(e_1), \\ F_2(E_2) &= f_{o_2}(f_{o_1}(e_1), e_2), \\ &\ldots \\ F_n(E_n) &= f_{o_n}(f_{o_{n-1}}(\ldots f_{o_1}(e_1), e_{n-1}), e_n) \\ &= \text{SQL} \end{aligned}$$

Further, LIST attempts to generate verifiable sub-queries based on the AST and dependencies, which is accomplished through grammar-based depth-first search (DFS) traversal. In a parsed AST, each node corresponds to an SQL operation and the leaf nodes represent basic elements such as table names or constants, as shown in Figure 1, and an executable sub-query could be represented as a subtree traversing from the root to the dependent values required by current operations, provided that higher-level operations depend on the lower-level elements or outputs.

Importantly, to ensure that each sub-SQL remains executable, we manually append the $f_{\text{select}}(\cdot, *)$ operation to the first $n-1$ sub-SQLs. We denote T as a set of sub-SQLs, expressed by:

$$T = (T_0, T_1, \ldots, T_n) = \text{LIST}(\text{SQL}) \quad (3)$$

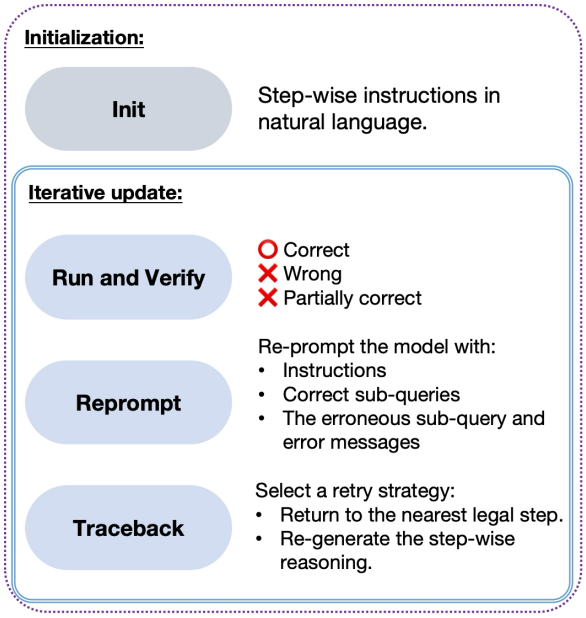

Figure 2: The four distinct states in the SQTS process.

LIST compares the dependencies of sibling nodes, with the execution order of parallel executions being fixed. For any nested execution structure, this step can be iteratively applied, ultimately identifying the minimal expansion steps. Detailed rules and coverage are presented in Appendix B.

Thus, single-hop reasoning is implemented as an iterative process of generating S_i with LIST, guided by the minimal yet valid expansion of keywords and elements.

In practice, LIST is built on top of SQLGlot [†], serving as our syntax tree generation and traversal tool, and is theoretically compatible with multiple dialects.

2.2 Regulated Step-Wise Generation with In-Context Learning

Inspired by Monte Carlo Tree Search (MCTS) (Brandfonbrener et al., 2024), we build up our SQL Grammar Tree-based Search (SQTS) for the constrained generation, which consists of four states: **Init**, **Run and Verify**, **Reprompt**, and **Traceback**, as described in Figure 2.

During the initialization phase, the model is provided with a set of examples that follow the single-hop reasoning paradigm to incrementally build SQL queries, only verbalizing the steps into natural language instructions. In practice, this fixed set of examples is generated by first decomposing SQL queries into sub-queries using LIST and then backtranslating them with GPT-4o, as described in Appendix C.1.

[†] https://github.com/tobymao/sqlglot

Then, SQTS formulates an iterative framework for the runtime step-wise verification. This process involves continuous interaction with the LIST and exploration of various sub-query generation strategies guided by the provided steps. (1) Each step instruction generates a corresponding SQL query, which is subsequently validated by the LIST decomposer and an external verifier. SQTS proceeds to the next step only upon successful validation of the current step, while cases identified as (partially) wrong trigger a Reprompt or Traceback process. (2) During the Reprompt process, the model is provided with the database schema, the question, the recorded correct sub-queries, and the erroneous sub-query along with its corresponding error message. This comprehensive feedback enhances the robustness of the generative model by offering specific error signals. (3) SQTS enters the traceback process after n unsuccessful attempts at a given step. In this process, SQTS either reverts to the most recent valid node or regenerates the step instructions if no valid state exists. This design minimizes the likelihood of ineffective iterations and enhances the overall efficiency of the query-build process.

In summary, SQTS algorithmically constrains the generation process, with the step-wise instructions and SQL sub-queries being collected and inserted by an algorithmic tool. The number of correction attempts for each node is set as `max_local_attempts`, and the total number of global generation attempts is set as `max_global_attempts`.

3 Preliminaries

3.1 Dataset and Metrics

The experiments adopt the recently released BIRD (Li et al., 2023) dataset, containing 12,751 text-to-SQL pairs across 95 databases, as a realistic and challenging test environment. We adopted the Execution Accuracy (EX), evaluating the correctness of SQL output by comparing the results of predicted queries with reference queries on specific database instances, as our primary evaluation metric.

3.2 Models and Prompts

Our experiments were carried out on Llama-3-8B-Instruct, its 70B version (Meta, 2024) and GPT-4o. We use Llama-3-8B and Llama-3-70B for short. All tasks relevant to Llama-3 models were performed on 8 NVIDIA A10 GPUs.

Methods	Training	BIRD-dev
w/ Schema Linking		
DIN-SQL(GPT-4)(Pourreza and Rafiei, 2023)	×	50.72%
DAIL-SQL(GPT-4)(Gao et al., 2023)	×	54.76%
MAC-SQL(GPT-3.5)(Wang et al., 2024)	×	50.56%
MAC-SQL(GPT-4)(Wang et al., 2024)	×	59.39%
SuperSQL(Li et al., 2024a)	×	58.60%
TA-SQL(Qu et al., 2024)	×	56.19%
w/o Schema Linking		
ChatGPT	×	37.22%
CodeS(15B, 5-shot)(Li et al., 2024c)	×	45.44%
GPT4	×	50.56%
DTS-SQL(Deepseek-7B)(Pourreza and Rafiei, 2024)	✓	55.80%
SEA-SQL(GPT-3.5+Mistral-7B)(Li et al., 2024b)	✓	56.13%
CodeS(15B, SFT)(Li et al., 2024c)	✓	58.47%
Ours (Llama-3-70B)	×	56.32%
Ours (GPT-4o)	×	**60.56%**

Table 1: Comparison of execution accuracy on the BIRD development set across different methods and models.

The prompts were intentionally and automatically constructed with fixed examples (ID=11,12,15,65, BIRD-train) and templates in Appendix C.

4 Experiments

We will compare the effectiveness of our method with mainstream approaches that with or without the schema linking module. Additionally, an ablation study is conducted to examine the impact of LIST decomposition and the verifier mode. We also present the coverage and cost analysis of LIST in Appendix. B.2 and E.

4.1 Main Results

We primarily validated the effectiveness of our method in an in-context learning manner on BIRD-dev dataset. We categorize trending methods into two folds, with or without schema linking module, as shown in Table 1 for comparison.

Our method demonstrates to be a novel and effective way for in-context learning without schema linking module. When schema linking modules are excluded, our method using GPT-4o achieves the best performance among open methods, including those with fine-tuning.

A demonstration of effectiveness with single reasoning path and at a comparable or better cost. Our method naturally demonstrated the results of step-wise syntactical validation along **one** reasoning path, since the step-wise instructions are generated only once. This suggests that methods employing multi-path reasoning and answer generation could hold greater potential, especially if more robust feedback signal tools or models can be developed.

4.2 Ablation Study

Furthermore, we conducted ablation tests. Due to cost considerations, these tests were performed on open-source models with few-shot prompting and WV (weak verifier), as shown in Table 2.

Methods	BIRD-dev Categories			
	easy	medium	challenging	all
Llama-3-8B	42.05%	20.04%	18.62%	33.18%
Ours-WV(8B)	42.70%	21.98%	15.17%	33.83%
Ours (8B)	**51.46%**	**27.16%**	**20.69%**	**41.20%**
Llama-3-70B	55.68%	36.64%	31.72%	47.72%
Ours-WV(70B)	62.05%	46.98%	32.41%	54.69%
Ours (70B)	**64.32%**	**50.43%**	**37.93%**	**56.32%**

Table 2: Our performance on BIRD-dev with open-source models under different settings.

Here, the weak verifier refers to a set-up where, after task decomposition, an SQL runtime environment is employed to provide feedback signals such as syntax errors or whether the query results are empty, without query decomposition. This feedback guides the model through the maximum of 3 iterations of error correction. Given that Llama-3 models have a context length limit of 8K tokens, we recorded each generation result along with the environment feedback and reorganized them into prompts for subsequent iterations.

Consistency of improvements across categories. Our approach yielded comparable gains across easy, medium, and challenging questions. Intuitively, leveraging reasoning capabilities more effectively, rather than simplifying the problem with external tools, is more likely to lead to balanced improvements.

The contribution is influenced by the model capabilities. We observed a steeper improvement curve on the 8B model compared to the 70B model. Ours-WV(8B) gains marginal benefit from solely step-wise reasoning prompts that without the step-wise verifier (0.65%), possibly due to limited instruction-following capabilities. However, when provided with step-wise verification, it can identify and rectify a fair amount of errors. The 70B model, on the other hand, achieves substantial improvement from step-wise reasoning alone (6.03%) and can further compensate for some generated grammatical errors through step-wise verification. The intuition behind could be that larger models possess stronger intrinsic reasoning abilities and can better leverage explicit step-wise prompts.

5 Conclusion

By decomposing the AST, LIST identifies a method to break down ontological SQL generation tasks into single-hop reasoning steps, while ensuring that each step is grammatically verifiable. Through a straightforward in-context learning approach, utilizing step-wise reasoning and step-wise generation, we achieved profound results using LLM inference alone. Furthermore, through ablation tests, LIST demonstrated the need and potential for integration with existing methods, which could shed some light on further RL-based methods.

6 Related Works

6.1 LMs for text-to-sql

The rationale behind Text-to-SQL pipeline can be abstracted into two primary steps: schema linking and generation (Popescu et al., 2003, 2004). A significant amount of methods focus on further decomposing this process and have proven effective. (1) Schema linking involves mapping natural language inputs to database relationships, mitigating the impact of the variability of natural language and handling the implicit relationships between database elements (Lei et al., 2020; He et al., 2019; Hong et al., 2024; Cao et al., 2021; Cai et al., 2022; Zhong et al., 2021). (2) Generation is regarded as the process of decoding the intermediate step into an executable SQL query. Existing methods focus on executing modules sequentially and algorithmically (Yu et al., 2018; Li et al., 2024a), such as using separate prompt templates for decomposed steps (Pourreza and Rafiei, 2023; Zhang et al., 2023; Dong et al., 2023) or combining LLMs with entity retrieval (Gao et al., 2023; Talaei et al., 2024), which we refer to as the modular approach. These independent schema linking modules imply that they must provide all relevant database elements, as any omission would inevitably result in errors (Maamari et al., 2024), unless models could be dynamically involved into each process. Based on this distinction, leveraging LLMs inherent strengths while integrating that isolated modularity into the reasoning and exploration processes represents a promising avenue for future development (Pourreza et al., 2024).

6.2 LLM for Reasoning

Reasoning with large language models (LLMs) involves breaking down complex inputs into a sequence of intermediate steps that lead to the final answer (Cobbe et al., 2021). This approach has been demonstrated with chain-of-thought (CoT) prompting (Wei et al., 2023), where the model processes information step by step.

We focus on the role of a controlled verifier in complex program reasoning. While existing methods that generate reasoning chains in a single step often encounter issues like error propagation (Chen et al., 2022), advancements such as self-consistency (Wang et al., 2022) and reasoning via planning (RAP) aim to mitigate these problems. Self-consistency employs majority voting over multiple reasoning chains to reduce errors, while RAP integrates Monte Carlo Tree Search (MCTS) simulations to enhance decision-making during reasoning (Hao et al., 2023).

7 Limitations

In order to achieve controlled and effective reasoning path exploration, our approach still has several limitations.

First, the validation methods used in this work are limited to simple in-context learning and the exploration of a single reasoning path. How to build more complex systems based on LIST, such as validation beyond syntax or constructing preferred reasoning paths, remains to be explored.

Second, the subqueries generated by LIST are not always perfectly verifiable. For example, two tables with only a `JOIN` but without an `ON` clause will be transformed by the engine into a Cartesian product, which can be extremely expensive. We have partially addressed this issue by methods such as reverse validation (starting from more complete SQL sub-queries) or setting resource limits and timeouts for each validation thread, achieving 100% coverage of syntactic decomposition on the BIRD dataset. However, this issue has not been tested in more complex scenarios and remains insufficiently resolved. This is crucial in explorative reasoning, as the verifier often becomes the limiting factor in performance (Cobbe et al., 2021). More comprehensive validation and further improvements are needed.

Lastly, although our approach demonstrates comparable or superior efficiency in cost analysis (Appendix E), expanding the entire paradigm to multi-path search inevitably leads to higher costs. Further exploration is needed to better construct external verifiers that collaborate with LLMs.

References

Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku.

David Brandfonbrener, Simon Henniger, Sibi Raja, Tarun Prasad, Chloe Loughridge, Federico Cassano, Sabrina Ruixin Hu, Jianang Yang, William E. Byrd, Robert Zinkov, and Nada Amin. 2024. Vermcts: Synthesizing multi-step programs using a verifier, a large language model, and tree search. *Preprint*, arXiv:2402.08147.

Hasan Alp Caferoğlu and Özgür Ulusoy. 2025. E-sql: Direct schema linking via question enrichment in text-to-sql. *Preprint*, arXiv:2409.16751.

Ruichu Cai, Jinjie Yuan, Boyan Xu, and Zhifeng Hao. 2022. Sadga: Structure-aware dual graph aggregation network for text-to-sql. *Preprint*, arXiv:2111.00653.

Ruisheng Cao, Lu Chen, Zhi Chen, Yanbin Zhao, Su Zhu, and Kai Yu. 2021. LGESQL: Line graph enhanced text-to-SQL model with mixed local and non-local relations. In *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)*, pages 2541–2555, Online. Association for Computational Linguistics.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *ArXiv*, abs/2110.14168.

Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, lu Chen, Jinshu Lin, and Dongfang Lou. 2023. C3: Zero-shot text-to-sql with chatgpt. *Preprint*, arXiv:2307.07306.

Ben Eyal, Amir Bachar, Ophir Haroche, Moran Mahabi, and Michael Elhadad. 2023. Semantic decomposition of question and sql for text-to-sql parsing. *Preprint*, arXiv:2310.13575.

Yujian Gan, Xinyun Chen, Qiuping Huang, and Matthew Purver. 2022. Measuring and improving compositional generalization in text-to-sql via component alignment. *Preprint*, arXiv:2205.02054.

Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R. Woodward, John Drake, and Qiaofu Zhang. 2021. Natural sql: Making sql easier to infer from natural language specifications. *Preprint*, arXiv:2109.05153.

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

Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards complex text-to-sql in cross-domain database with intermediate representation. *Preprint*, arXiv:1905.08205.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*.

Pengcheng He, Yi Mao, Kaushik Chakrabarti, and Weizhu Chen. 2019. X-sql: reinforce schema representation with context. *Preprint*, arXiv:1908.08113.

Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2024. Next-generation database interfaces: A survey of llm-based text-to-sql. *Preprint*, arXiv:2406.08426.

Wenqiang Lei, Weixin Wang, Zhixin Ma, Tian Gan, Wei Lu, Min-Yen Kan, and Tat-Seng Chua. 2020. Re-examining the role of schema linking in text-to-SQL. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6943–6954, Online. Association for Computational Linguistics.

Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024a. The dawn of natural language to sql: Are we fully ready? *Proceedings of the VLDB Endowment*, 17(11):3318–3331.

Chaofan Li, Yingxia Shao, and Zheng Liu. 2024b. Sea-sql: Semantic-enhanced text-to-sql with adaptive refinement. *Preprint*, arXiv:2408.04919.

Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024c. Codes: Towards building open-source language models for text-to-sql. *Preprint*, arXiv:2402.16347.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Preprint*, arXiv:2305.03111.

Karime Maamari, Fadhil Abubaker, Daniel Jaroslawicz, and Amine Mhedhbi. 2024. The death of schema linking? text-to-sql in the age of well-reasoned language models. *Preprint*, arXiv:2408.07702.

Meta. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

OpenAI. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Ana-Maria Popescu, Alex Armanasu, Oren Etzioni, David Ko, and Alexander Yates. 2004. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 141–147, Geneva, Switzerland. COLING.

Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th International Conference on Intelligent User Interfaces*, IUI '03, page 149–157, New York, NY, USA. Association for Computing Machinery.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O. Arik. 2024. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql. *Preprint*, arXiv:2410.01943.

Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Preprint*, arXiv:2304.11015.

Mohammadreza Pourreza and Davood Rafiei. 2024. Dts-sql: Decomposed text-to-sql with small large language models. *Preprint*, arXiv:2402.01117.

Ge Qu, Jinyang Li, Bowen Li, Bowen Qin, Nan Huo, Chenhao Ma, and Reynold Cheng. 2024. Before generation, align it! a novel and effective strategy for mitigating hallucinations in text-to-sql generation. *Preprint*, arXiv:2405.15307.

Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. Chess: Contextual harnessing for efficient sql synthesis. *Preprint*, arXiv:2405.16755.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2024. Mac-sql: A multi-agent collaborative framework for text-to-sql. *Preprint*, arXiv:2312.11242.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. *Preprint*, arXiv:2201.11903.

Sohee Yang, Elena Gribovskaya, Nora Kassner, Mor Geva, and Sebastian Riedel. 2024. Do large language models latently perform multi-hop reasoning? *Preprint*, arXiv:2402.16837.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Preprint*, arXiv:2305.10601.

Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. 2018. Syntaxsqlnet: Syntax tree networks for complex and cross-domaintext-to-sql task. *Preprint*, arXiv:1810.05237.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2019. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *Preprint*, arXiv:1809.08887.

Hanchong Zhang, Ruisheng Cao, Lu Chen, Hongshen Xu, and Kai Yu. 2023. Act-sql: In-context learning for text-to-sql with automatically-generated chain-of-thought. *Preprint*, arXiv:2310.17342.

Victor Zhong, Mike Lewis, Sida I. Wang, and Luke Zettlemoyer. 2021. Grounded adaptation for zero-shot executable semantic parsing. *Preprint*, arXiv:2009.07396.

A Hyperparameters

While the baseline experiments set `temperature=0`, our approach sets `temperature=0.8` and `top_p=0.95` to allow multi-round self-correction. We set `max_local_attempts` as 3 and `max_global_attempts` as 15.

B LIST

B.1 Formula

Given a natural language query Q and database structure information DB (including mapping relationships, columns, tables, etc.), the goal is to input Q and DB into a large language model (LLM). The LLM will reason to derive the corresponding SQL query Y.

Let the input $I = \{Q, DB\}$, then the LLM will output a series of steps S, which can be represented as:

$$S = \{S_1, S_2, \ldots, S_n\} = R(I)$$

The SQL statement Y_i corresponding to each step S_i can be represented as:

$$Y_i = \begin{cases} R_i(I, S_i) & \text{if } i = 1 \\ R_i(I, S_i, Y_{1:i-1}) & \text{if } i \in [2, \ldots, n] \end{cases}$$

By generating SQL Y_i for each step, we can obtain the final answer $Y_n = Y$ at the last step.

If Y_i runs with an error during reasoning, we use LIST to split the SQL into multiple executable sub-SQLs T, verify them individually and provide the verification information to the model to correct the erroneous SQL.

Before giving the mathematical expression of LIST, let's consider an example for better understanding of LIST.

Suppose we have the following SQL: `SELECT cols FROM t1 JOIN t2 WHERE cons`.

According to the execution order of SQL, we can represent this SQL as:

$$\begin{aligned}\text{SQL} = f_{\text{select}}(&\\ &f_{\text{where}}(f_{\text{join}}(f_{\text{from}}(\mathtt{t_1}, \mathtt{t_2}), \mathtt{cons}), \mathtt{cols})\\ )&\end{aligned}$$

Define $o = \{\text{select, join, from, } \dots\}$, which includes database operations such as selection, joining, and specifying data sources. Define $e = \{\text{cols, cons, tables, } \dots\}$ which e represents various database elements such as columns (cols), constraints (cons), and tables. Then,

$$\text{SQL} = f_{o_n}(f_{o_{n-1}}(\dots f_{o_1}(e_1), e_{n-1}), e_n)$$

Now, we can represent the nested structure of the function $f(\cdot)$ in the following more concise manner:

$$\begin{aligned}F_1(E_1) &= f_{o_1}(e_1),\\ F_2(E_2) &= f_{o_2}(f_{o_1}(e_1), e_2),\\ &\dots\\ F_n(E_n) &= f_{o_n}(f_{o_{n-1}}(\dots f_{o_1}(e_1), e_{n-1}), e_n)\\ &= \text{SQL}\end{aligned}$$

Our LIST method originates from the innermost nested structure of the function, incrementally adding an operator and its corresponding entity at each step until the full SQL statement is reconstructed. Importantly, to ensure that each sub-SQL remains executable, we manually append the $f_{\text{select}}(\cdot, *)$ operation to the first $n-1$ sub-SQLs. We denote T as a set of sub-SQLs, expressed by:

$$T = (T_0, T_1, \dots, T_n) = \text{LIST}(\text{SQL})$$

where each T_i for $i = 0, 1, \dots, n$ is defined as follows:

$$\begin{aligned}T_0 &= f_{\text{select}}(f_{\text{from}}(\mathtt{t_1}), *) = f_{\text{select}}(F_1(E_1), *),\\ T_1 &= f_{\text{select}}(F_2(E_2), *),\\ &\dots\\ T_{n-1} &= f_{\text{select}}(F_n(E_n), *),\\ T_n &= f_{\text{select}}(F_n(E_n), \mathtt{cols}) = F_n(E_n).\end{aligned}$$

In summary,

$$T_i = \begin{cases} f_{\text{select}}(F_{i+1}(E_{i+1}), *), & i \in [0, n-1],\\ F_i(E_i), & i = n.\end{cases}$$

Thus, we obtain the mathematical expression of LIST.

The SQL syntax tree can be converted into an execution order table based on its dependencies and handed over to the SQL engine core for execution. The syntax tree is composed of nested expression objects, such as those shown in Figure 1, where each expression object's key and its child key form parent-child node relationships. Similar to other tree structures, the syntax tree can be serialized through various algorithms, such as infix and postfix expressions. Based on this intuition, we employ an SDFS (SQL Depth First Search) method to traverse the SQL.

In our algorithm, each exploration step can be written as E, where the explored keywords can be seen as an operation $f_{\text{operation}_i}$, and the included database elements can be regarded as reasoning R for the problem Q, database DB, and mapping relationship M, forming an input I; i.e.,

$$I = R(Q, DB, M)$$

SQL also has advanced constructs, such as CTE (Common Table Expressions), which are identified as Table Expressions, and various dialects like ClickHouse and Hive, which offer different syntax variations. However, since the syntax tree is cross-linguistic, our approach is applicable to the decomposition of different SQL variants.

B.2 Coverage

We tested the coverage of LIST on BIRD-dev, dividing it into three metrics: split pass rate (SPR), complete pass rate (CPR), and sub-query pass rate (SQPR). For complete SQL statements, we denote the total number of SQLs tested as sql_num, the number of split-pass SQLs as $split\_pass$, and the number of SQLs with no syntax errors across all clauses as sql_pass. At the clause level, sub_query_num and sub_query_pass represent the total number of split clauses and the number of clauses without syntax errors, respectively. For

BIRD-dev, $sql\_num = 1534$.

$$SPR = \frac{split\_pass}{sql\_num}$$
$$CPR = \frac{sql\_pass}{sql\_num}$$
$$SQPR = \frac{sub\_query\_pass}{sub\_query\_num}$$

Metric	Pass Rate
Split Pass Rate	1.0
Complete Pass Rate	0.9485
Sub-query Pass Rate	0.9842

Table 3: Pass Rates for LIST on BIRD-dev.

C Prompt Templates

C.1 Prompt for Generating Step-Wise Reasoning

Start

```
## Instruction
You will be given a database schema
and a question-answer pair.
The question is about the database
schema, the answer, however, is a
partial step-by-step solution to the
question. Your task is to generate
natural language translation of the
partial steps.
## Database Schema\n
{schema}
## Question\n
{question}
## Answer\n
{LIST results}

## Your translation and reasoning pro-
cess that shows why you need to do
this with the reference to the question
and database schema\n

example: [[Step 1: content]]
```

C.2 Prompt for Our Method

Start

```
## Instruction:
You are a data scientist specializing
in text-to-SQL tasks. You should write
a valid SQLite to solve the following
question based on the database scheme
and hint.

## Database Schema:
{schema}

## Question:
You need to decompose the following
question into several operation steps:
{question}

Quote your step-by-step instruction
in [[]]. Do not generate SQL.

## Steps:
```

Init

```
## Schema:
{Schema}

## Instruction:
Revisit the schema, question, and
finish the step in SQLite grammar.
Answer with Question analysis,
Step analysis, Schema analysis and
Result in sql block.

## Question:
{Question}

## Step:
{Steps}
```

Running

```
## Schema:
{Schema}

## Instruction:
Revisit the schema, question, and
accomplish the SQL generation in
SQLite grammar. Answer with Question
analysis, Step analysis, Schema
```

```
analysis and Result in sql block.

## Question:
{Question}

## Step:
{Steps}

## Verified Code:
{Node_Content}

Return the final sql in ```sql ```
format.
```

Reprompt

```
## Schema
{Schema}

## Instruction
Answer with Question analysis, Step
analysis, Schema analysis and Result
in sql block.

## Question:
{Question}

## Step:
{Current_Step}

## Error need to be handled:
Generated SQL:
```sql\n{Node_Content}\n```
SQLs that run well:
```sql\n{Node_Runnable_Code}\n```
When it turns to:
```sql\n{Node_Error_Code}\n```
error occured.
Return the final sql in ```sql ```
format.
```

C.3 Prompt for Naive Few-Shot

Reprompt

```
You are a data scientist special-
izing in text-to-SQL tasks.
You should write a valid SQLite
to solve the following question
based on the database schema and
hint.
### Database schema:
{schema}
### Question:
{question}
###SQL:
```

D Case Study

We randomly chose 12 samples to do case study, which ids in [80, 104, 422, 439, 536, 707, 840, 849, 929, 953, 1025, 1061].

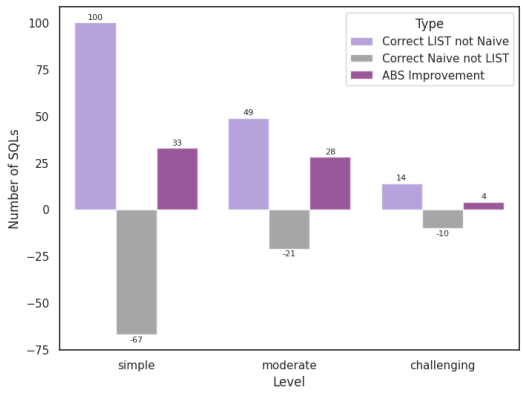

Figure 3: Case study on our method (70B, 1-shot).

Error Type	Llama3-8b-instruct	Llama3-70b-instruct
redundant columns	707, 840, 929, 1024	104, 707, 929, 1024
over-complication	80, 422, 894, 1061	80, 422, 439, 894, 1061
timeouts	104, 953	953
type errors	536	536
other	439	840

Table 4: Case study details.
```

Method	Turns	Input tokens	Output tokens
TA-SQL(Qu et al., 2024)	4	8,540,001	-
DIN-SQL(Pourreza and Rafiei, 2023)	4	33,203,612	-
CHESS(Low budget)(Talaei et al., 2024)	6	33,761,389	-
E-SQL(Caferoğlu and Özgür Ulusoy, 2025)	3	56,090,710	1,162,772
Ours(Llama-3-8B)	8.7973	28,177,560	1,767,875
Ours(Llama-3-70B)	6.1121	19,539,863	1,203,187
Ours(GPT-4o)	5.8370	18,449,648	1,114,580

Table 5: Cost analysis details.

E Cost Analysis

In the following cost analysis, we consider several metrics as standards: the number of dialogue turns, the number of input tokens, and the number of output tokens. We focus on comparisons with schema linking-based methods, as both involve multiple interactions with LLMs.

For comparability, we additionally calculate the average number of dialogue turns for our method (total iterative turns / number of questions) and compare it with the fixed number of dialogue turns in schema linking-based methods. For token counts, we use OpenAI's tiktoken library for tokenization across all models to ensure consistency.

Results are shown in Table 5. Based on the pricing of GPT-4o ($2.5 per 1M input tokens and $10 per 1M output tokens), the running cost of Ours(GPT-4o) is approximately $55, with an average cost of $0.035 per question. We also observed that using more powerful models for inference reduces the likelihood of falling into ineffective attempts. Specifically, when comparing the number of questions that exhausted the $max\_global\_attempts$ limit of 15, the results were as follows: GPT-4o: 54; Llama-70B: 86; Llama-8B: 280.

For comparison, we have the experiments and details as follows:

- **DIN-SQL**: The scope of our analysis includes 'schema_linking_prompt', 'classification_prompt', 'easy_prompt', and 'correction_prompt'. We excluded 'medium_prompt' and 'hard_prompt' as the classification module was not actually executed. Also, instead of providing actual input content in the templates, we used placeholders such as 'schema_links' and 'sql_query'.

- **E-SQL**: Referring to the cost analysis in this work, we calculate the token consumptions by multiplying the number of samples in the BIRD dataset.

- **TA-SQL**: For efficiency, we reproduce and simulate the pipeline by inserting a fixed dummy SQL. The whole process contains the column meaning, dummy prompts, SR prompts and SR2SQL prompts.

- **CHESS**: The same to the TA-SQL, we reproduce the keyword extraction, column filtering, table selection, column selection, candidate generation and revision process according to the CHESS (IR, SS, CG) configuration with low budget.

In summary, we found that our approach of verification coupled with exploration improves efficiency as the model's capabilities increase. Intuitively, stronger models possess better reasoning and instruction-following abilities, which is why our method can achieve approximately a 35% efficiency gain over Llama-3-8B.

The schema linking paradigm requires the use of a large number of few-shot examples or additional modules, such as question enrichment, to enhance the robustness of SQL as a formal language. While the schema linking paradigm remains effective, there is always a trade-off between cost, inference efficiency, and accuracy that needs to be considered.

Error type	Examples	Ours(8B)	Ours(70B)
no such column	"id": 2, "sql": " SELECT * FROM frpm ORDER BY FRPM Count DESC ", "status": -1, "message": "Error: no such column: FRPMCount"	3662	547
Got null result	"id": 0, "sql": " SELECT * FROM (SELECT "Free Meal Count (Ages 5-17)" / "Enrollment (Ages 5-17)" AS eligible_free_rate FROM frpm WHERE "Educational Option Type" = 'Continuation') AS subquery ORDER BY eligible_free_rate ASC LIMIT 3; ", "status": -1, "message": "Warning: Got null result."	823	380
Wrong syntax for SQLite	"status": -1, "id": 0, "sql": " SELECT CDSCode FROM frpm WHERE (Enrollment (K-12) + Enrollment (Ages 5-17)) > 500; ", "message": "Wrong syntax for SQLite. splitter syntax error"	314	70
Ambiguous column name	"id": 0, "sql": " SELECT COUNT(raceId) FROM races r JOIN results res ON r.raceId = res.raceId WHERE r.year = 2008 AND r.name = 'Australian Grand Prix'; ", "status": -1, "message": "Error: ambiguous column name: raceId"	64	24

Table 6: Cost analysis details.

F Error Analysis

In our work, we found that (1) the issues encountered during multi-turn iterations largely fall into these existing categories, and (2) problems unique to multi-turn dialogue scenarios, such as repeated failures after multiple modifications, are particularly challenging to classify into specific categories, given that the model may produce different errors during each modification attempt.

Therefore, during our experiments, we primarily focused on the error types reported by the verifier and conducted a comprehensive analysis of the validation results across all iterations of the model.

The following are the four most frequently occurring cases listed in Table 6:

- No such columns: This occurs when an incorrect column name is used or when a column is sought from the wrong table.
- Got null result: This happens due to incorrect conditions or when the correct answer is an empty result. In practice, we only report a warning rather than an error for such cases.
- Wrong syntax for SQLite: Since the benchmark uses SQLite as the execution environment, we restrict LIST to generating SQL statements that conform to SQLite's syntax rules.
- Ambiguous column name: This arises when columns with identical names exist across multiple tables, requiring column names to be explicitly specified.