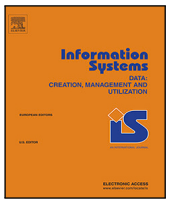
Information Systems 138 (2026) 102670

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is



HLR-SQL: Human-like reasoning for Text-to-SQL with the human in the loop☆

Timo Eckmann [a,*], Matthias Urban [a], Jan-Micha Bodensohn [a,b], Carsten Binnig [a,b]

[a] Technical University of Darmstadt, Hochschulstraße 10, Darmstadt, 64289, Hesse, Germany
[b] German Research Center for Artificial Intelligence (DFKI), Landwehrstraße 50A, Darmstadt, 64293, Hesse, Germany

ARTICLE INFO

Keywords:
Text-to-SQL
NL2SQL
Natural language interface for databases
Reasoning

ABSTRACT

Recent LLM-based approaches have achieved impressive results on Text-to-SQL benchmarks such as Spider and Bird. However, these benchmarks do not accurately reflect the complexity typically encountered in real-world enterprise scenarios, where queries often span multiple tables. In this paper, we introduce HLR-SQL, a new approach designed to handle such complex enterprise SQL queries. Unlike existing methods, HLR-SQL imitates Human-Like Reasoning with LLMs by incrementally composing queries through a sequence of intermediate steps, gradually building up to the full query. This is an extended version of Eckmann et al. (2025). The new contributions are centered around incorporating human feedback directly into the reasoning process of HLR-SQL. We evaluate HLR-SQL on a newly constructed benchmark, Spider-HJ, which systematically increases query complexity by splitting tables in the original Spider dataset to raise the average join count needed by queries. Our experiments show that state-of-the-art models experience up to a 70% drop in execution accuracy on Spider-HJ, while HLR-SQL achieves a 9.51% improvement over the best existing approaches on the Spider leaderboard. Finally, we extended HLR-SQL to incorporate human feedback directly into the reasoning process by allowing the LLM to selectively ask for human help when faced with ambiguity or execution errors. We demonstrate that including the human in the loop in this way yields significantly higher accuracy, particularly for complex queries.

1. Introduction

LLMs dominate Text-to-SQL. Text-to-SQL, the task of translating natural language questions into SQL statements, has recently gained traction because it enables non-expert users to query databases, substantially enlarging their user bases. Recent state-of-the-art approaches employ Large Language Models (LLMs) with prompt engineering [1–4] or supervised fine-tuning [5] and achieve impressive translation accuracies on established academic benchmarks such as Spider [6] and Bird [7]. For example, DAIL-SQL [2], currently the best publicly listed approach on Spider, can correctly translate 86.6% of queries in the Spider test set.

Real-world queries are more complex. Existing benchmarks, however, do not fully reflect the complexity of real-world settings. For instance, as reported in [8], real-world databases are often highly normalized, causing even simple natural language questions to require 4.01 joins per query on average. In contrast, queries in the Spider benchmark [6] require only 0.5 joins on average, and those in Bird [7] only around 1 join (i.e., 2 tables). More importantly, only 60 out of 7000 queries in Spider and 74 in Bird (approximately 0.85% and 0.78% of the respective training sets) require four or more joins. This low maximum join count demonstrates that existing benchmarks do not fully capture the real world with SQL queries that involve many join operations.

Text-to-SQL for complex queries. In this paper, we thus set out the goal of studying Text-to-SQL in scenarios that require a high number of join clauses. As we show in our evaluation, existing approaches leading classical benchmarks fail on such complex queries. To solve them, we argue that Text-to-SQL systems must approach the task of SQL generation more like humans do. When a human is tasked with formulating a complex SQL statement, they typically do not write the whole statement in one go. Instead, they start with a simpler statement like *SELECT * FROM <tablename>*, which they might even execute. Based on this initial statement, we then iteratively make it more complex or combine multiple simple SQL statements using joins or nested queries.

Human-like reasoning for Text-to-SQL. Based on this idea, we propose a novel approach to emulate human reasoning when solving the Text-to-SQL task through iterative composition, which involves actions such as note-taking, query refinement, database exploration, and error correction. In this paper, we present a first prototype called HLR-SQL,

☆ This article is part of a Special issue entitled: 'HILDA' published in Information Systems.

* Corresponding author.
E-mail address: timo.eckmann@tu-darmstadt.de (T. Eckmann).

https://doi.org/10.1016/j.is.2025.102670
Received 28 July 2025; Received in revised form 16 December 2025; Accepted 19 December 2025
Available online 2 January 2026

which implements this idea on top of an LLM that uses an external memory to keep track of the artifacts of the incremental query construction. An important aspect of HLR-SQL is that rather than following a fixed procedure, it operates with complete autonomy and may adopt different actions to incrementally refine a query. As such, HLR-SQL autonomously revises and combines queries from previous steps until it determines that a satisfactory solution has been found. Finally, we show that the procedure of iteratively composing SQL queries in HLR-SQL can be easily extended to include a human in the loop. Whenever the LLM encounters difficulties, such as an ambiguous question or errors in its SQL execution or due to a large number of tables to join, HLR-SQL can choose to ask a human for help. As we show in our evaluation, this leads to substantially higher translation accuracies, especially when generating complex SQL queries with many joins. Moreover, we show that HLR-SQL asks for help more often when the query is complex and only rarely when it is simple, involving the human only when necessary.

A more realistic benchmark is needed. To better understand the challenges arising from more complex queries with higher join counts for the Text-to-SQL setting, we propose *Spider-HJ*, a novel dataset derived from Spider [6]. Unlike Spider, where queries require only 0.5 joins on average, and only 60 queries require four or more joins, *Spider-HJ* features queries with an average of 5.64 joins per query and a maximum of 20 joins. To obtain such a dataset, we keep the high-quality natural language questions of Spider and modify only the database schema in a way that requires more joins to answer the questions. Thus, any performance decrease can be directly attributed to the increased complexity of the underlying schema. In this way, *Spider-HJ* enables a precise assessment of a system's resilience to an increase in necessary join operations.

Initial results. Our initial evaluation shows that increased query complexity leads to a sharp decline in the execution accuracy of current state-of-the-art approaches. For example, DAIL-SQL [2], which is top-ranked on the Spider leaderboard, drops from 86.6% on the Spider test set to just 15.14% on our dataset. In contrast, HLR-SQL significantly improves over these approaches in accuracy by 9.51% without human in the loop and by 22.62% with human in the loop, highlighting the benefits of user feedback for HLR-SQL.

2. Human-like reasoning for Text-to-SQL

In this section, we present our vision of Text-to-SQL pipelines that reason like humans do. Furthermore, we present our initial prototype HLR-SQL that mimics this Human-Like Reasoning as presented in [9].

2.1. The need for a new approach

Modern Text-to-SQL approaches like DIN-SQL [3], DAIL-SQL [2], and MAC-SQL [10] achieve strong performance but suffer from a fundamental limitation: they typically rely on a fixed, predefined sequence of generation steps. This restriction contrasts sharply with how humans naturally use multiple steps to solve complex tasks. In the case of Text-to-SQL, humans craft complex queries by starting with simple queries, iteratively refining them, observing feedback from the database by executing these simpler queries, and then incrementally improving them. Naturally, this process will take more iterations for complex SQL statements and thus be longer than for simple statements. Especially queries involving multiple joins and nested sub-queries over large-scale databases with many tables benefit from such an iterative, human-like approach that automatically adapts the amount of "effort" (i.e., compute and number of database interactions) to the complexity of the question at hand.

Some recent approaches, such as CHASE-SQL [4] and XiYan-SQL [11], increase the amount of effort independently of the question complexity by employing ensemble methods that generate multiple sample solutions and then choose a final query using a specially tuned selection model. MAC-SQL [10], on the other hand, tries to first estimate question complexity and then decomposes questions accordingly into multiple sub-questions. These sub-questions are then translated independently into SQL sub-queries and combined to obtain the final SQL. Unlike HLR-SQL, they do not incrementally build queries and test sub-queries by executing them. As such, wrong assumptions or mistakes early in the reasoning chain (i.e., in the formulation of early sub-queries) might lead to error propagation, resulting in final SQL statements that are vastly different from the ground truth and, thus, hard to fix.

For example, consider the query "Find the name of instructors who taught in Fall 2009 but not in Spring 2010" in a database where the instructor name is stored in an *instructors* table, and the *courses* table stores information about when they taught. A potential first sub-question could be "Which instructor taught in Fall 2009?" which already involves a join and can thus fail when the wrong join key is used. In this case, MAC-SQL will be oblivious to the mistake as it does not check sub-queries. In contrast, a human would first verify the list of fall instructors by executing a respective SQL query on the database and fixing it until it works, then separately identifying those in Spring, and finally combining the two results in a larger query, thereby reducing the likelihood of such error propagation.

2.2. Our approach: HLR-SQL

HLR-SQL tries to mimic the human-like, iterative problem-solving approach to SQL query composition. Instead of committing to a single static pipeline, an LLM agent refines queries based on intermediate feedback, splitting a complex problem into smaller tasks and revising incorrect assumptions repeatedly along the way. We call this cycle the *draft, test, learn, and improve* process.

Key aspects of HLR-SQL. Most importantly, the agent in HLR-SQL is completely *autonomous* in its iterative Text-to-SQL construction. This means that it can terminate or continue depending on its assessment of the problem's complexity and the quality of its current solution. Nevertheless, to prevent endless loops, HLR-SQL can only re-iterate until a maximum of iterations is reached. Another key aspect of HLR-SQL is its *memory*, which tracks all information about previous steps and sub-results that have been obtained so far. To achieve complete autonomy, in each step, we provide the LLM with the question, the available data, and its memory from previous steps and let it decide which next steps it wants to take.

Composition process in HLR-SQL. Fig. 1 shows the behavior of HLR-SQL using a simple example. Denoted as ①, the user submits a natural language question to the agent, which triggers the incremental query construction. In Fig. 1, there is already a previous SQL sub-query present in the memory, which resulted from a previous iteration. In step ②, HLR-SQL picks one of four capabilities to refine this query: *(1) Formulate Sub-Question in NL:* The LLM formulates a sub-question in natural language by breaking down the input question into sub-question(s) similar to chain-of-thought prompting [12]. *(2) Generate or combine SQL sub-queries:* Here, the LLM either generates completely new SQL fragments or uses the existing SQL sub-queries from memory and combines them with a new sub-query, which it generates. *(3) Formulate final SQL statement:* The LLM chooses this capability if it intends to create the final SQL query that solves the original Text-to-SQL problem it was presented with. *(4) Acknowledge final SQL:* As a last capability, HLR-SQL also provides the option to acknowledge the final SQL query. This step terminates the overall procedure.

If the agent selects capability (2) or (3), the agent will output a SQL string and execute it over the database (see ③+④ in Fig. 1). A sample of the query execution result is put into the memory of the agent for further refinement. In our example in Fig. 1, the LLM chooses

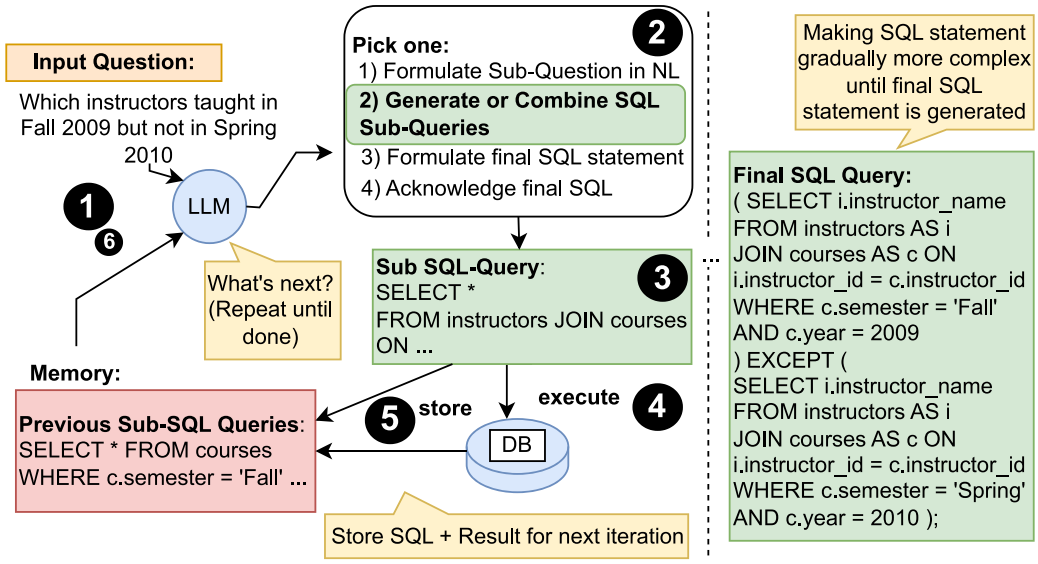


Fig. 1. HLR-SQL creates SQL statements by iteratively combining and revising SQL sub-queries. The procedure starts by putting the input question and the content of the memory (initially empty) in a prompt ①. Note that the figure shows a later iteration where a sub-SQL from a previous iteration is already in the memory. Given the prompt, the LLM decides ② to combine that existing query with a new query, which it generates ③. The generated query is run on the database ④, and the result is stored in the memory ⑤. This process of iterative refinement repeats ⑥ until the LLM acknowledges the final SQL query (right side) to be correct.

to combine the existing SQL sub-query with a new sub-query that joins the *instructors* and *courses* tables, which is then executed on the actual database (step ④) and the result stored in the memory (step ⑤). After multiple iterations, HLR-SQL will then output and acknowledge the results of the final SQL query (see right side of Fig. 1).

Autonomy and error recovery. A benefit of HLR-SQL is the self-reliance of the agent when exploring different parts of the problem as it sees fit. This allows it to fix assumption errors that go beyond simple SQL syntax errors. An example of this might be a synonym mismatch where the user asks for "films", but the column is labeled "movies". Our agent can test both columns and adapt based on the obtained sub-results.

Memory management. As discussed earlier, the memory in HLR-SQL is a simple json file that extends through interaction rounds. It is not managed by the LLM or the system as long as the next prompt does not exceed the context length of the LLM. If it does happen to exceed the context length the earliest sub-results are trimmed and sub-questions are trimmed until it fits into context. To reduce the load on the context even further, we also include only up 10 rows of the sub-results in the memory json.

3. Human-like reasoning with the human in the loop

Even when a SQL query is constructed incrementally, essential information may still be missing or ambiguous. For example, if a user asks for customer information but the database contains multiple tables with customer data, it is unclear which table to use without further clarification. In such cases, user input is necessary to disambiguate the query. To address this, prior Text-to-SQL approaches have introduced mechanisms to collect user feedback. However, if user interaction is triggered too frequently, it can become intrusive and disrupt the overall experience. Our approach addresses this limitation by integrating user feedback as one of several reasoning-time actions. Rather than always prompting the user, HLR-SQL decides — based on its internal reasoning — whether user input would help resolve uncertainty and, if so, what form that input should take.

This design is motivated by the observation that HLR-SQL may not be able to recover from certain errors, especially when the natural language query lacks critical information or when the schema is highly complex (e.g., involving many joins). In such cases, human assistance becomes essential. Just as a developer struggling with a SQL query might ask a more knowledgeable colleague for help — especially if that colleague is more familiar with the data or can resolve ambiguities in the question — HLR-SQL can consult the user when needed. By selectively involving the human only when necessary, our method enables more accurate and efficient query generation while maintaining a streamlined user experience.

Integrating the human into HLR-SQL. As depicted in Fig. 2, we integrate human interaction into HLR-SQL by adding a new action that the LLM can choose from (see ② in Fig. 2). While this action can be chosen at any time, in principle, we want to minimize the number of unnecessary interaction requests by HLR-SQL. Hence, we instructed the LLM to choose this action after an SQL execution error or if it detects a possible ambiguity in the given question.

When HLR-SQL picks the action to ask the user for help, it must formulate a clarifying question that is displayed to the user (see ③). For this, we allow the LLM to generate a question to the user which might be most helpful. In the example in Fig. 2, HLR-SQL is unsure whether external instructors should be included in the result. With the additional action, it now has the possibility to clarify that by asking the user. The user responds in plain natural text (*yes* in the example) that HLR-SQL stores in its memory (see ④), thereby resolving the difficulties for the next iteration.

In the example, HLR-SQL now knows from the user feedback that external instructors should be included in the result and can now generate the final SQL query accordingly. Our evaluation shows that this way of integrating human interaction into the Text-To-SQL translation leads to substantial gains in translation accuracy. Moreover, our approach leads to little human interaction for simple queries and more human interaction for complex queries.

4. Spider-HJ: A benchmark with many joins

Existing benchmarks are great resources for advancing Text-to-SQL research [13,14]. Yet, they so far have less of a focus on complex queries that require many joins (see Table 1). In this paper, we thus present a new benchmark called *Spider-HJ*, which builds on Spider [6] but contains queries with increasing SQL complexity in terms of the

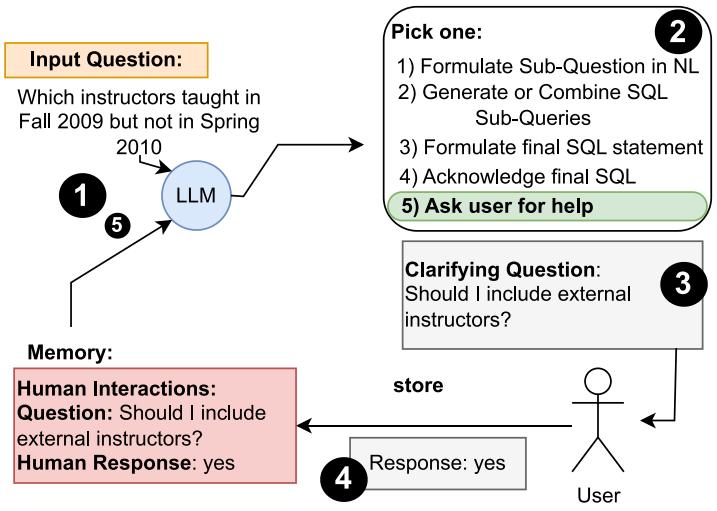


Fig. 2. HLR-SQL with the Human in the Ldoop. Adding an action that allows HLR-SQL to ask the user for help when it encounters challenges due to high complexity or ambiguity in the question or the schema ②. When this action is selected by HLR-SQL, it generates a clarifying question ③ and stores the user feedback in its memory ④. This effectively allows HLR-SQL to make progress in its next iteration ⑤.

number of joins. We leverage the existing NL-to-SQL pairs from the Spider dataset and introduce schema modifications that necessitate additional join operations. As such, we can accurately reason that any performance differences are due to the increase in the number of required joins and the schema complexity.

Constructing a dataset with many joins. To increase the total number of joins required for a given question, we adopt a multi-stage process that transforms the original database schema from Spider along with the SQL queries into several variants all corresponding to the same natural language question. Each of these variants has an increased number of joins. For example, if a query selects columns c_1, c_2 of table T, we create three variants: one that splits off c_1 (introducing one extra join), one that splits off c_2 using another join key (introducing one extra join), and one that splits off both columns from the original table (introducing two extra joins).

The steps of our benchmark construction process are as follows: *(1) Parsing the original SQL query:* Each SQL query is first parsed to identify the specific columns used in its execution. This step ensures that splitting these columns necessitates joins. *(2) Partitioning and augmenting the used columns:* The columns identified are separated into new tables. For each column, a corresponding table is created. This process can lead to an unrealistically large number of small tables that only contain foreign keys and the queried column. To make sure that the resulting tables have a realistic number of columns, we append a random number of columns (between 3–5) that are generated by an LLM. *(3) Random column selection and processing:* Since we only split columns that are required by the query into new tables, this makes it very obvious which tables must be joined. Therefore, we randomly select up to 5 additional columns from the original schema — even if they are not directly referenced by the query — and subject them to a similar partitioning process. These columns are also allocated to new tables and padded with three to five LLM-generated columns, thereby making it less obvious for an LLM that split tables are needed in the golden query. *(4) Generating multiple schema variants:* For each base SQL query, we systematically generate all possible combinations of column splits. This results in multiple schema variants corresponding to the same natural language question, thereby increasing the number of required join operations and introducing nuanced variations in the database schema. *(5) Finally, for each resulting schema variant, we adapt the SQL query accordingly and validate that the query output on the new schema is identical to the original SQL query executed on the original database.* We repeat this process to incrementally add more joins. We argue that the schema-splitting approach in Spider-HJ creates semi-realistic database schemas because it reflects how real-world databases evolve through normalization—distributing related attributes across multiple tables without changing the underlying query intent. Crucially, the semantics of the benchmark queries remain unchanged: the natural language question and its logical SQL meaning are identical; however, the schema now requires additional joins to retrieve the same information.

Concrete example. A concrete splitting example can be seen in Fig. 3. The original Spider dataset provides a question, a golden SQL query, and n tables. Out of those n tables, exactly two are referenced in the golden SQL and thus relevant for splitting. In this case, these are the *purchases* and the *products* table. The golden query references two columns in total (one in the projection and one in the where-condition). As we can split off each of those columns individually or both together, there are 3 possible splits. For each possible split, a new database is created that will only be used for this single input question. The first splitting opportunity is to split off the *amount* column into a new table that can be joined with the original purchases table through a newly created pk/fk relation ①. The name of the new table, along with 3–5 new columns and up to 10 data values for each column, is generated through an LLM call. Furthermore, 3–5 other columns will be randomly chosen and split in the same way to make the database more realistic and the task more challenging. Finally, we update the SQL query to include the additional join. In fact, since the columns to split are decided based on the golden SQL query, it is guaranteed that we introduce an additional join condition. Afterwards, we continue with the other two splitting opportunities in the same way. In the example, the procedure results in three new Text-To-SQL data points, each with a different golden SQL query and database, but with the same input question.

Characteristics of Spider-HJ. By adopting this approach, we generate more than 20,000 questions with an average join count of 5.64 joins per question. As shown in Table 1, this is more than 10 times the average joins per question of Spider and more than 5 times in comparison to Bird. A benefit of this approach is that we can make use of the quality of the existing natural language questions of Spider, only increasing the complexity of the schema and SQL query without changing the question complexity.

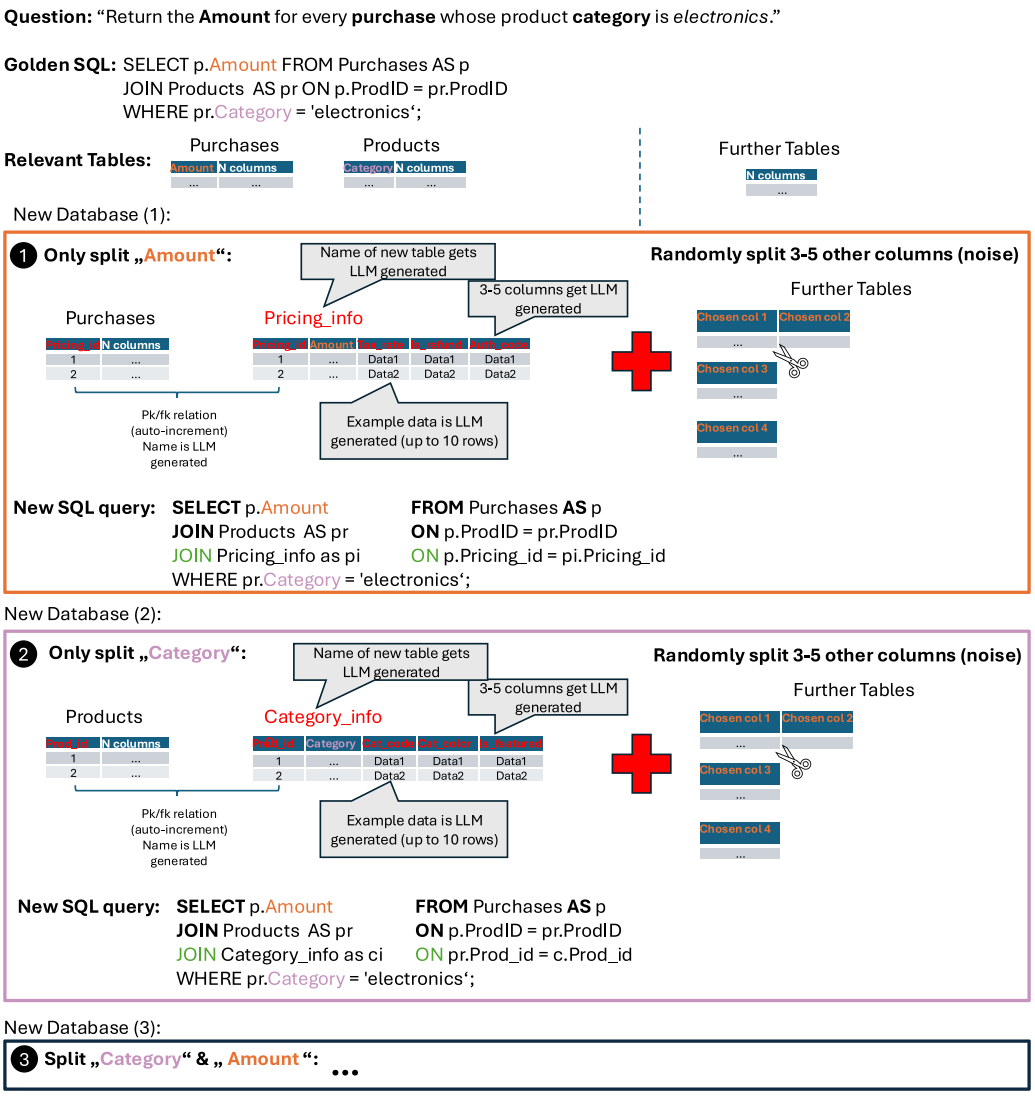


Fig. 3. Spider-HJ is created by analyzing the golden SQL and identifying the mentioned columns and tables (here Products.Category and Purchases.Amount are the columns mentioned). For each mentioned column, we create a new database by splitting it off, forcing a new join in the gold SQL. In ①, the column *Amount* is split from the *Purchases* table. It is moved into a newly created table that we name and augment using an LLM. To make the database more realistic, we also split off 3–5 other columns in the same way. After we split off both mentioned columns individually, we create another database where we split off both columns, adding two new joins to the SQL query. Thus, in this case, we created three new Text-To-SQL data points, each with a different SQL query and database, but with the same question.

Table 1
Comparison of Spider, Bird, and *Spider-HJ* regarding join counts. Our dataset contains 10 times more joins on average than Spider and also a 2.5 times increase in the maximum join counts found in the benchmark.

Dataset	Avg. # joins	Max. # joins
Spider	0.54	8
Bird	1.02	6
***Spider-HJ* (ours)**	**5.64**	**20**

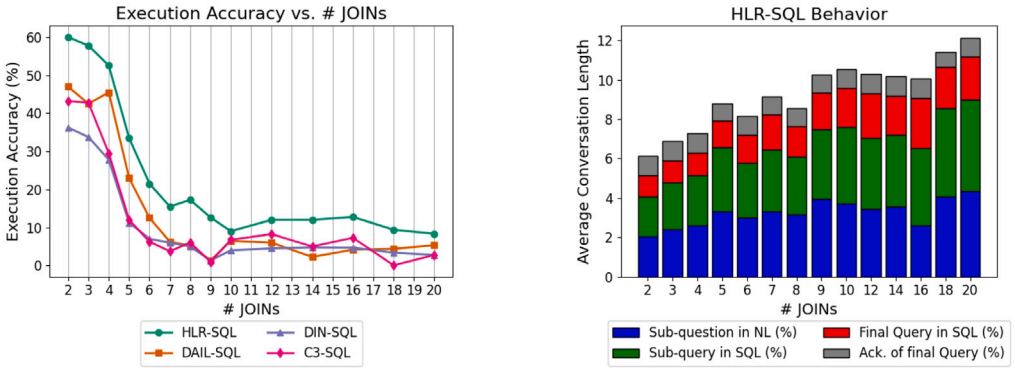
5. Initial experimental evaluation

In this section, we present our initial evaluation of HLR-SQL on *Spider-HJ*. Our initial results indicate that it can better adapt to the more complex schemata than other recent Text-to-SQL approaches.

5.1. Experiment setup

Dataset. To examine the effect of varying the number of joins, we use the previously-introduced *Spider-HJ* dataset and randomly sample 400 distinct queries for the different join counts present in *Spider-HJ*. For the join counts 16, 18, and 20, there are only 381, 301, 120 questions in Spider-HJ. Therefore, we include all of them. Additionally, we evaluated both approaches on BEAVER to see how well it performs in a real-world setting.

Metric. In line with other Text-to-SQL research, we employ execution accuracy (**EX**) as our primary metric. A generated query is considered correct if its execution result matches that of the golden SQL query, regardless of differences in SQL query formulation. In particular, our implementation of execution accuracy considers two output tables as identical even if the columns are in a different order, as long as their



(a) HLR-SQL vs Baselines (b) HLR-SQL: Number of Steps

Fig. 4. Subplot (a) shows the accuracy on *Spider-HJ*. Subplot (b) shows the amount and type of iterative steps of HLR-SQL.

contents are identical. For row order, we analyze the golden SQL query: If the query string contains an ORDER BY clause, we are strict and only count the output as correct if the order of rows matches. Otherwise, two query outputs are considered identical even if the row order is different. Finally, we are also strict when it comes to duplicates: If the output of the predicted SQL contains duplicates and the golden query does not, it is considered incorrect.

Models. We use **GPT-4o-Mini-2024-07-18** because it is considerably more cost-efficient than GPT-4o while still demonstrating remarkable performance on the original Spider test set [15].

Baselines. As baselines, we use the top three publicly available and listed approaches on the Spider leaderboard [6]: **DIN-SQL** [3], **DAIL-SQL** [2], and **C3-SQL** [1]. We slightly adapted all approaches to work with GPT-4o-Mini-2024-07-18. These modifications ensure that the comparisons are fair by using the same LLM for all methods. In Experiments 1 and 2, we run HLR-SQL without human interaction. Experiments 3 and 4 then show how human interaction can further improve execution accuracy.

HLR-SQL. Since HLR-SQL can autonomously continue querying indefinitely, we impose an upper limit of 25 iterative steps. At the 25th step, HLR-SQL is required to make its final guess, and no further steps are allowed. Additionally, we limit the size of sub-results from sub-queries to 10 rows to not exceed the context limit.

5.2. Exp. 1: Varying #Joins

In this experiment, we analyze the accuracies of all baselines and HLR-SQL on the *Spider-HJ* dataset. We break down the results by the number of joins in the SQL query ranging from 2–20 joins. The results of the baselines and HLR-SQL are shown in Fig. 4(a). On the x-axis, we display the number of joins, increasing from left to right. On the y-axis, we display the different execution accuracies.

As shown in Fig. 4, the execution accuracy of all baselines decreases significantly with increasing joins counts. This shows the difficulty of state-of-the-art approaches to adapt to the more complex schema. Additionally, we can see that HLR-SQL (green) continuously outperforms all baselines. However, it also shows the same behavior overall, indicating that high join counts are more challenging in general.

5.3. Exp. 2: Conversation length

Since humans take more iterative steps to come to a final SQL for more complex queries, we expect HLR-SQL to do the same. To understand the actual behavior of HLR-SQL, we analyze the conversation length (number of iterative prompts) for each join count, as shown in Fig. 4(b). As can be seen, the average conversation length generally increases with the number of joins. Furthermore, the unexpected performance increase from 7 to 8 joins is also represented in the behavior of HLR-SQL, as we can see a decrease in the average conversation length. This leads us to believe that there are indeed additional characteristics that make these questions easier, even though their number of join operations is higher. The same can be seen for the performance increase from 10 to 16 joins. From this, we conclude that the LLM correctly uses its given capabilities more for harder questions, which aligns with human behavior. However, the overall performance shows that for highly complex queries, further improvements are necessary to fully imitate human behavior and human performance.

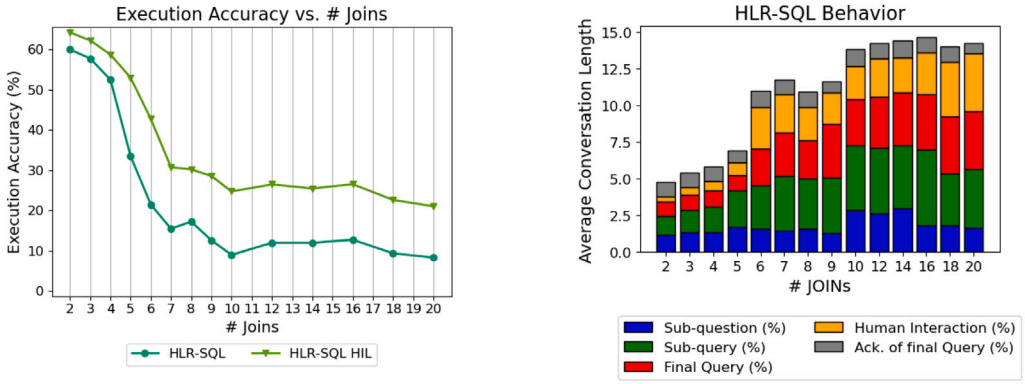
5.4. Exp. 3: Human interaction

Finally, as explained in Section 3, we also experiment with enabling human interaction as a way for HLR-SQL to resolve difficulties it encounters during translation. However, evaluating HLR-SQL on all questions of the previous experiments comes with infeasible manual overhead, since human interaction can be triggered multiple times per query.

Thus, we decide to instead emulate the human using a second LLM similar to [16]. To emulate the human, we picked the same model as for HLR-SQL (GPT-4o-Mini-2024-07-18). To ensure that the model that emulates the human is able to answer the question, we follow [16] and provide it with the ground-truth SQL to guide HLR-SQL to the correct solution. Thus, it can simply compare the question with the ground-truth to answer it. To avoid the LLM which emulates the human is leaking the ground truth SQL, we tuned its prompt accordingly and added another LLM call to double-check its response. However, we found that the emulated human never tried to leak the ground truth SQL to HLR-SQL.

Fig. 5 shows the results. In terms of execution accuracy (Fig. 5(a)), we see a significant improvement compared to the approaches without human interaction. Importantly, the improvements are more pronounced for complex queries with high join counts since these queries provide more difficulties during translation in identifying the required tables and join paths as well as attributes. For high join counts, human feedback can lead to improvements of up to 15% while for simple queries the gains are lower (e.g., for 2 joins the gains are approximately 5%).

Looking at HLR-SQL's behavior in terms of conversations lengths (see Fig. 5(b), orange portion of bars), we see that it asks the human for help more often when the query is more complex. For small numbers of joins, human interaction is picked only for a small fraction of queries. However, for larger join counts, HLR-SQL often has to ask the user multiple times until it is confident to generate the final query. While



(a) Execution Accuracy of HLR-SQL with human in the loop (HLR-SQL HIL) vs Baselines.

(b) Number of Steps with human in the loop

Fig. 5. Human interaction leads to consistent improvements of over 10% over HLR-SQL without human interaction for queries with high join count (left). HLR-SQL only asks the human for help when it cannot translate the query on its own, leading to a higher number of interactions for queries with many joins (right).

Table 2
Execution accuracy (%) across different join counts. HLR-SQL-HIL represents our human-in-the-loop approach, while HLR-SQL does not include the human-in-the-loop.

Join	HLR-SQL-HIL	HLR-SQL	DAIL-SQL	C3-SQL	DIN-SQL
2	**64.2**	60.0	47.0	43.2	36.2
3	**62.2**	57.8	42.5	42.8	33.8
4	**58.8**	52.5	45.5	29.5	27.8
5	**53.0**	33.5	23.0	12.0	11.2
6	**42.8**	21.5	12.5	6.2	7.0
7	**30.8**	15.5	6.2	3.8	6.0
8	**30.2**	17.2	5.2	6.0	5.0
9	**28.5**	12.5	1.2	1.0	1.2
10	**24.8**	9.0	6.5	6.8	4.0
12	**26.5**	12.0	6.0	8.2	4.5
14	**25.5**	12.0	2.2	5.0	4.8
16	**26.5**	12.9	4.2	7.3	4.7
18	**22.6**	9.3	4.3	0.3	3.3
20	**20.8**	8.3	5.0	2.5	2.5

this leads to additional overhead by the user who has to answer the clarifying questions, the improved execution accuracy is clearly worth it, as we have shown before. These results show that human interaction is a great way to resolve these difficulties with queries with many joins.

To enable a better comparison, the concrete performance numbers are provided in Table 2

5.5. Exp. 4: Real-world performance

To test the performance of both approaches on a real-world dataset we used BEAVER [8]. As shown in Table 3, the performance of HLR-SQL and HLR-SQL-HIL is impressive, although we use a cheaper model than what was tested in BEAVER. By iteratively constructing the final query, it is possible to achieve an execution accuracy of 10.84%. If a human is included in the loop, execution accuracy improves to 18.23%, highlighting the benefits of human interaction. However, BEAVER remains a challenging benchmark for Text-To-SQL approaches.

6. Discussion on human interaction

A key question for human-in-the-loop Text-to-SQL systems is what kind of user interacts with the system and what the user must know to give useful feedback. For example, if users have no SQL or domain knowledge, they cannot help the system with SQL syntax or column abbreviations. However, even if they are not well-versed in SQL, they may still be effective helpers if they understand the logic of the question. This section analyzes what kinds of questions appear during interaction in HLR-SQL-HIL, and in particular, how many are answerable without SQL knowledge.

Table 3
HLR-SQL and HLR-SQL-HIL perform improve greatly on BEAVER [8]. While the original publication reported 0% execution accuracy, HRL-SQL (without human interaction) and HRL-SQL-HIL both improve execution accuracy, with HRL-SQL-HIL achieving the highest at 18.23%.

Approach	Model	Execution accuracy
Reported in BEAVER [8]	gpt-4o	0.0%
HLR-SQL	gpt-4.0-mini	10.84%
HLR-SQL-HIL	**gpt-4.0-mini**	**18.23%**

6.1. Method

In experiment 3, a total of 14.483 questions were asked by HLR-SQL-HIL to the user. Out of those, we randomly sampled 1446 questions from all join counts in 20 batches and classified them using an LLM-based classifier, leading to 72 categories. However, many of them were overlapping, such as “Join Error Help” and “SQL Join Help”. Therefore, the 72 initial categories were consolidated into three non-overlapping categories through iterative refinement using an LLM. The consolidated categories were then manually checked by randomly sampling 50 questions per category. All 1446 questions were then re-classified into the consolidated categories. These are: “Schema Knowledge Gap”, “Domain/Business Logic Gap”, and “SQL Technical Gap”. In the following, we first discuss each category and afterwards analyze which answers help the system most.

6.2. Question categories

Schema knowledge gap. These are questions that ask about table names, column meanings, or relationships between tables. For example: “In which column is the ‘genre’ stored?” To answer these questions, the user requires a basic understanding of the data and access to schema documentation.

Domain/Business logic gap. These are questions that ask about valid data values or business rules. For example: “Can an order have more than one shipping address?” These require domain knowledge but not SQL expertise.

Table 4
Categorization of clarifying questions across all query complexity levels (N = 1446). The majority of questions (78.2%) can be answered by domain experts without SQL knowledge, requiring only access to schema documentation or domain expertise.

Category	Count	SQL-Expertise?
Schema knowledge gap	1001 (69.2%)	No
Domain/Business logic gap	130 (9.0%)	No
SQL technical gap	315 (21.8%)	Yes
Answerable w/o SQL expertise	**1131 (78.2%)**	

Table 5
Helpfulness and answer speeds by question type for all 1446 questions. "High" and "Med." indicate estimated helpfulness of the generated questions; "Time" indicates how fast the questions can be answered (<30 s = Fast, >2 min = Slow).

Category	Count	High	Med.	Time
Schema knowledge gap	1001 (69.2%)	33.7%	66.3%	Fast
Domain/Business logic gap	130 (9.0%)	0%	100%	Fast
SQL technical gap	315 (21.8%)	100%	0%	Slow

SQL technical gap. Questions that fall under this category are those that explicitly ask about SQL syntax or query structure. Example: "How should I structure the JOIN conditions?" These require advanced SQL knowledge to answer correctly.

6.3. Results

As shown in Table 4, most questions (69.2%) fall into the Schema Knowledge Gap category. This indicates that most interaction effort focuses on resolving unclear or complex schema structures in Spider-HJ. Only 9.0% of questions fall under the Domain/Business Logic Gap category. Their low share suggests that Spider-HJ mainly tests schema reasoning rather than the need for domain knowledge, which aligns with the classification of Spider in [7]. The remaining 21.8% of questions belong to the SQL Technical Gap category. They represent cases where the system's SQL generation process runs into errors, such as a missing column, and requests guidance from users. Since much of the challenge from Spider-HJ stems from the increase in join conditions, we expect the majority of errors in this category to be related to joins. In fact, all 50 manually checked questions were related to a wrong join condition.

All in all, 78.2% of all questions can be answered without SQL knowledge according to our LLM-based classification. This indicates that most human interaction relies on users who are familiar with the data structure rather than actual SQL syntax, with only a smaller subset requiring the users to have SQL expertise.

However, while this makes it easier for non-experts to answer the questions, we also observed that sometimes the LLM formulates schema-related questions, although other types of questions would be better suited. In particular, we originally expected that increasing the number of tables would primarily result in more SQL-specific clarifications on how to write join conditions. In fact, our manual review suggests that the added join complexity often results in missing join conditions and related execution errors. For instance, the column *flightnumber* may be mistakenly assumed to exist in the *airlines* table instead of being correctly joined with the *flights* table, which causes a "no such column" error in SQLite, as this column does not exist in the *airlines* table. However, in such cases, the system requests schema information from the human (e.g., where the flight number is stored), even though the real issue is a wrong join (joining with airlines instead of flights).

What type of questions improve system accuracy. The three question types, classified using an LLM, differ not only in who can answer them but also in their contribution to improving SQL generation. Therefore, we prompted an LLM to classify the positive impact of each generated question on the reasoning process as "Medium" or "High". Importantly, we provide the LLM with the entire reasoning chain as context so that it can accurately assess whether each question was helpful or not. Similarly, we also predict for each generated question how long a user would need to answer it. Table 5 shows the results. Schema Knowledge Gap questions are predicted to have medium improvements overall, likely because they address fundamental ambiguities during early query construction. While clarifying a column name helps, the system must still generate correct JOIN conditions, WHERE clauses, and aggregations. Domain/Business Logic Gap questions occur infrequently, suggesting the natural language interface successfully captures most business logic without explicit clarification. Therefore, these kinds of questions are also classified to yield medium improvements. Predictably, SQL Technical Gap questions exhibit the highest improvements (but also demand the most expertise). This occurs because they represent final syntax corrections after the system has already identified correct tables, columns, and relationships, making each clarification the "last mile" toward a complete query. Nevertheless, their complexity restricts accessibility to users with specialized knowledge.

7. The road ahead

In this paper, we examined the impact of complex SQL queries on the Text-to-SQL task. Our experiments have shown that current state-of-the-art approaches have focused too much on queries with low join counts (i.e., less than 4 joins), which is far from realistic scenarios. We argue that in order to solve these queries, a new human-like approach is necessary. Our initial prototype HLR-SQL shows substantial improvements over existing approaches for these queries by mimicking human reasoning and iteratively generating and testing more and more complex queries. However, many open questions remain.

Real-world data and other dimensions of complexity. While a high number of joins is certainly a major driver of complexity in real-world SQL queries, they can also be complex due to a variety of other reasons. For instance, queries can be nested or contain complex selection, filter, or group-by expressions. Moreover, real-world data can also be complex, for example by containing ambiguous or cryptic column names and table values. In the future, we thus intend to evaluate our approach on newer datasets such as BEAVER [8] and Spider 2.0 [17] that incorporate these complexities.

Text-to-SQL as a reasoning task. Despite the superior performance of HLR-SQL on complex queries of *Spider-HJ*, there are still many queries where it fails, especially with high join counts. Thus, an in-depth error analysis is necessary to understand the failure points of HLR-SQL and when they diverge from human behavior. Moreover, with the announcement of OpenAI o1 and Deepseek R1 [18], reinforcement learning trained reasoning models have shown promising results for reasoning tasks such as mathematical problem solving [18]. Following our argument in this paper, we think that these can be a very good fit for Text-to-SQL as well. However, currently, these models cannot interact with the database during reasoning like HLR-SQL can do, which is crucial to mimic human behavior and generate complex SQL statements, as we have shown in this paper.

A database interface designed for LLMs. So far, the interface between HLR-SQL and the database is that HLR-SQL generates SQL sub-queries that are executed on the database. However, for many human-like subtasks, SQL on its own is too limited. For example, assuming we want to select all courses in fall, we need to know how *fall* is represented in the database, which could be *fall*, *autumn*, or even a numerical value. This requires a method to look for semantically similar values in the

database. Recent approaches solve these issues before SQL generation during schema linking, where they find potentially relevant values from the database and include them in the Text-to-SQL prompt. However, this again results in a static Text-to-SQL pipeline that starts with schema linking and ends with SQL generation. In contrast, we think that schema linking should be part of the reasoning process, and finding values in the database should be realized by extending the database interface for the LLM with a keyword search to search for values semantically similar to "fall". While parallel work [19] already explores this direction to some extent, we argue that this idea can be extended further by including interfaces that can search through metadata and documentation during the reasoning process.

CRediT authorship contribution statement

Timo Eckmann: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Matthias Urban:** Writing – review & editing, Writing – original draft, Project administration, Methodology, Formal analysis, Data curation. **Jan-Micha Bodensohn:** Writing – review & editing, Writing – original draft, Validation, Software, Methodology. **Carsten Binnig:** Writing – review & editing, Supervision, Methodology, Investigation, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been supported by the Bundesministerium für Bildung und Forschung and the state of Hesse as part of the NHR Program and the HMWK cluster project 3AI. It was also partially funded by the LOEWE Spitzenprofessur of the state of Hesse. We also thank SAP, DFKI Darmstadt and hessian. AI for their support. Furthermore, it has benefited from the early stages of the funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), Germany under Germany's Excellence Strategy— EXC-3057; funding will begin in 2026.

References

[1] X. Dong, C. Zhang, Y. Ge, Y. Mao, Y. Gao, l. Chen, J. Lin, D. Lou, C3: Zero-shot text-to-SQL with ChatGPT, 2023, arXiv:2307.07306. URL https://arxiv.org/abs/2307.07306.

[2] D. Gao, H. Wang, Y. Li, X. Sun, Y. Qian, B. Ding, J. Zhou, Text-to-SQL empowered by large language models: A benchmark evaluation, 2023, arXiv:2308.15363. URL https://arxiv.org/abs/2308.15363.

[3] M. Pourreza, D. Rafiei, DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction, 2023, arXiv:2304.11015. URL https://arxiv.org/abs/2304.11015.

[4] M. Pourreza, H. Li, R. Sun, Y. Chung, S. Talaei, G.T. Kakkar, Y. Gan, A. Saberi, F. Ozcan, S.O. Arik, CHASE-SQL: Multi-path reasoning and preference optimized candidate selection in text-to-SQL, 2024, arXiv:2410.01943. URL https://arxiv.org/abs/2410.01943.

[5] H. Li, J. Zhang, H. Liu, J. Fan, X. Zhang, J. Zhu, R. Wei, H. Pan, C. Li, H. Chen, Codes: Towards building open-source language models for text-to-SQL, 2024, arXiv:2402.16347. URL https://arxiv.org/abs/2402.16347.

[6] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, D. Radev, Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task, 2019, arXiv:1809.08887. URL https://arxiv.org/abs/1809.08887.

[7] J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Cao, R. Geng, N. Huo, X. Zhou, C. Ma, G. Li, K.C.C. Chang, F. Huang, R. Cheng, Y. Li, Can LLM already serve as a database interface? A BIg bench for large-scale database grounded text-to-SQLs, 2023, arXiv:2305.03111. URL https://arxiv.org/abs/2305.03111.

[8] P.B. Chen, F. Wenz, Y. Zhang, D. Yang, J. Choi, N. Tatbul, M. Cafarella, Çağatay Demiralp, M. Stonebraker, BEAVER: An enterprise benchmark for text-to-SQL, 2025, arXiv:2409.02038. URL https://arxiv.org/abs/2409.02038.

[9] T. Eckmann, M. Urban, J.-M. Bodensohn, C. Binnig, HLR-SQL: Human-like reasoning for text-to-SQL, in: Proceedings of the Workshop on Novel Optimizations for Visionary AI Systems, NOVAS '25, Association for Computing Machinery, New York, NY, USA, 2025.

[10] B. Wang, C. Ren, J. Yang, X. Liang, J. Bai, L. Chai, Z. Yan, Q.-W. Zhang, D. Yin, X. Sun, Z. Li, MAC-SQL: A multi-agent collaborative framework for text-to-SQL, 2025, arXiv:2312.11242. URL https://arxiv.org/abs/2312.11242.

[11] Y. Gao, Y. Liu, X. Li, X. Shi, Y. Zhu, Y. Wang, S. Li, W. Li, Y. Hong, Z. Luo, J. Gao, L. Mou, Y. Li, A preview of XiYan-SQL: A multi-generator ensemble framework for text-to-SQL, 2025, arXiv:2411.08599. URL https://arxiv.org/abs/2411.08599.

[12] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, D. Zhou, Chain-of-thought prompting elicits reasoning in large language models, 2023, arXiv:2201.11903. URL https://arxiv.org/abs/2201.11903.

[13] X. Deng, A.H. Awadallah, C. Meek, O. Polozov, H. Sun, M. Richardson, Structure-grounded pretraining for text-to-SQL, in: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Association for Computational Linguistics, 2021, http://dx.doi.org/10.18653/v1/2021.naacl-main.105, URL http://dx.doi.org/10.18653/v1/2021.naacl-main.105.

[14] Y. Gan, X. Chen, Q. Huang, M. Purver, J.R. Woodward, J. Xie, P. Huang, Towards robustness of text-to-SQL models against synonym substitution, 2021, arXiv:2106.01065. URL https://arxiv.org/abs/2106.01065.

[15] H. Li, S. Wu, X. Zhang, X. Huang, J. Zhang, F. Jiang, S. Wang, T. Zhang, J. Chen, R. Shi, H. Chen, C. Li, OmniSQL: Synthesizing high-quality text-to-SQL data at scale, 2025, arXiv:2503.02240. URL https://arxiv.org/abs/2503.02240.

[16] M. Urban, J. Ding, D. Kernert, K. Vaidya, T. Kraska, Utilizing past user feedback for more accurate text-to-SQL, in: Proceedings of the Workshop on Human-in-the-Loop Data Analytics, HILDA '25, Association for Computing Machinery, New York, NY, USA, 2025, http://dx.doi.org/10.1145/3736733.3736739.

[17] F. Lei, J. Chen, Y. Ye, R. Cao, D. Shin, H. Su, Z. Suo, H. Gao, W. Hu, P. Yin, V. Zhong, C. Xiong, R. Sun, Q. Liu, S. Wang, T. Yu, Spider 2.0: Evaluating language models on real-world enterprise text-to-SQL workflows, 2025, arXiv:2411.07763. URL https://arxiv.org/abs/2411.07763.

[18] DeepSeek-AI, D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, X. Zhang, X. Yu, Y. Wu, Z.F. Wu, Z. Gou, Z. Shao, Z. Li, Z. Gao, A. Liu, B. Xue, B. Wang, B. Wu, B. Feng, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, D. Dai, D. Chen, D. Ji, E. Li, F. Lin, F. Dai, F. Luo, G. Hao, G. Chen, G. Li, H. Zhang, H. Bao, H. Xu, H. Wang, H. Ding, H. Xin, H. Gao, H. Qu, H. Li, J. Guo, J. Li, J. Wang, J. Chen, J. Yuan, J. Qiu, J. Li, J.L. Cai, J. Ni, J. Liang, J. Chen, K. Dong, K. Hu, K. Gao, K. Guan, K. Huang, K. Yu, L. Wang, L. Zhang, L. Zhao, L. Wang, L. Zhang, L. Xu, L. Xia, M. Zhang, M. Zhang, M. Tang, M. Li, M. Wang, M. Li, N. Tian, P. Huang, P. Zhang, Q. Wang, Q. Chen, Q. Du, R. Ge, R. Zhang, R. Pan, R. Wang, R.J. Chen, R.L. Jin, R. Chen, S. Lu, S. Zhou, S. Chen, S. Ye, S. Wang, S. Yu, S. Zhou, S. Pan, S.S. Li, S. Zhou, S. Wu, S. Ye, T. Yun, T. Pei, T. Sun, T. Wang, W. Zeng, W. Zhao, W. Liu, W. Liang, W. Gao, W. Yu, W. Zhang, W.L. Xiao, W. An, X. Liu, X. Wang, X. Chen, X. Nie, X. Cheng, X. Liu, X. Xie, X. Liu, X. Yang, X. Li, X. Su, X. Lin, X.Q. Li, X. Jin, X. Shen, X. Chen, X. Sun, X. Wang, X. Song, X. Zhou, X. Wang, X. Shan, Y.K. Li, Y.Q. Wang, Y.X. Wei, Y. Zhang, Y. Xu, Y. Li, Y. Zhao, Y. Sun, Y. Wang, Y. Yu, Y. Zhang, Y. Shi, Y. Xiong, Y. He, Y. Piao, Y. Wang, Y. Tan, Y. Ma, Y. Liu, Y. Guo, Y. Ou, Y. Wang, Y. Gong, Y. Zou, Y. He, Y. Xiong, Y. Luo, Y. You, Y. Liu, Y. Zhou, Y.X. Zhu, Y. Xu, Y. Huang, Y. Li, Y. Zheng, Y. Zhu, Y. Ma, Y. Tang, Y. Zha, Y. Yan, Z.Z. Ren, Z. Ren, Z. Sha, Z. Fu, Z. Xu, Z. Xie, Z. Zhang, Z. Hao, Z. Ma, Z. Yan, Z. Wu, Z. Gu, Z. Zhu, Z. Liu, Z. Li, Z. Xie, Z. Song, Z. Pan, Z. Huang, Z. Xu, Z. Zhang, Z. Zhang, DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning, 2025, arXiv:2501.12948. URL https://arxiv.org/abs/2501.12948.

[19] G. Xiong, J. Bao, H. Jiang, Y. Song, W. Zhao, Interactive-T2S: Multi-turn interactions for text-to-SQL with large language models, 2024, http://dx.doi.org/10.48550/ARXIV.2408.11062, CoRR abs/2408.11062. arXiv:2408.11062.