Odin: A NL2SQL Recommender to Handle Schema Ambiguity

Kapil Vaidya[1*], Abishek Sankararaman[2],
Jialin Ding[2], Chuan Lei[2], Xiao Qin[2], Balakrishnan Narayanaswamy[2], Tim Kraska[2]
[1]Parallel Web Systems [2]Amazon Web Services

ABSTRACT

NL2SQL (natural language to SQL) systems translate natural language into SQL queries, allowing users with no technical background to interact with databases and create tools like reports or visualizations. While recent advancements in large language models (LLMs) have significantly improved NL2SQL accuracy, schema ambiguity remains a major challenge in enterprise environments with complex schemas, where multiple tables and columns with semantically similar names often co-exist. To address schema ambiguity, we introduce Odin, a NL2SQL recommendation engine. Instead of producing a single SQL query given a natural language question, Odin generates a set of potential SQL queries by accounting for different interpretations of ambiguous schema components. Odin dynamically adjusts the number of suggestions based on the level of ambiguity, and Odin learns from user feedback to personalize future SQL query recommendations. Our evaluation shows that Odin improves the likelihood of generating the correct SQL query by 1.5–2× compared to baselines.

1 INTRODUCTION

NL2SQL (natural language to SQL) systems translate natural language questions into SQL queries, allowing users with no technical background to interact with databases and create tools like reports or visualizations. For example, a manager could ask, "*How many new customers did we acquire this quarter?*," and the system would generate the corresponding SQL query, execute it, and return the results in an accessible format. This broadens access to data-driven insights, making NL2SQL an essential tool for decision-making. Large language models (LLMs) have played a transformative role in advancing NL2SQL technology. Their ability to adapt to unfamiliar tasks by leveraging contextual examples allows them to generate accurate outputs. This adaptability has positioned LLM-based solutions at the top of various well-known NL2SQL benchmarks like Spider and BIRD [19, 36].

In enterprise environments, database schemas are often large and highly complex due to the integration of multiple data sources, frequent schema versioning, and table transformations. This complexity introduces ambiguity when translating natural language questions into SQL queries. Specifically, tables and columns with similar names can exist, making it difficult to identify the user's intended reference. We refer to this challenge as *schema ambiguity*, where multiple SQL queries, each using different tables or columns, could potentially be correct.

For instance, a user might ask, "*What are the average salaries by department?*", but the database schema could contain both a `curr_dept` table and a `dept_2022` table. In this scenario, it is unclear which table the user intends to query, leading to multiple potential SQL queries. Similarly, ambiguity can arise when dealing with column names. For example, when a user asks, "*What were the total sales last quarter?*", the `sales` table may have columns named both `gross_sales` and `net_sales`, either of which could be valid depending on the user's intent.

Furthermore, schema ambiguity can substantially impact the structure of a SQL query. For example, consider the question, "*What is the revenue per customer?*". In this case, the database schema might include both a `customers` table and an `orders` table, where the `orders` table contains a column named `revenue`, while the `customers` table may include an aggregated column, `total_revenue`. Depending on which table the user intends to reference, two structurally different SQL queries can be generated:

```
SQL1: SELECT customers.customer_id, SUM(orders.revenue)
FROM customers
JOIN orders ON customers.customer_id = orders.customer_id
GROUP BY customers.customer_id;
```

```
SQL2: SELECT customer_id, total_revenue
FROM customers;
```

SQL1 involves a join and aggregation across multiple rows in the `orders` table, while SQL2 simply retrieves pre-aggregated data from the `customers` table. This example demonstrates that schema ambiguity can have an effect on the complexity and runtime performance of the SQL query, making its resolution a critical aspect of NL2SQL systems.

Among multiple potential SQL queries, users often prefer one query over the others due to their preferences toward specific tables or columns in the schema. These preferences may reflect the user's familiarity with certain data sources or their expectations regarding the relevance of certain schema components. For instance, the user may prefer *gross_sales* instead *net_sales* for revenue analysis due to the business logic of their organization. By learning these preferences, a NL2SQL system could provide more personalized query suggestions, enhancing the user experience.

One approach to address schema ambiguity is to generate a set of SQL queries that accounts for all possible interpretations of the ambiguous schema components. This allows users to review the options and select the query that best fits their needs. However, generating such a set is challenging. LLM-based NL2SQL systems often struggle to produce a sufficiently diverse set of potential SQL queries, even when using techniques such as sampling methods or temperature adjustments, as highlighted by [2]. Additionally, the set cannot be too large, as it would overwhelm the user and make selecting the correct query difficult. The complexity increases further when users have specific preferences for tables or columns which need to be captured and incorporated into future questions.

To address the challenges presented by schema ambiguity, we introduce Odin, a NL2SQL recommendation engine that helps users manage and resolve schema ambiguity in NL2SQL tasks. Instead of returning a single SQL query, Odin presents a set of potential

[*]Work performed while employed at AWS.

queries for users to choose from. The number of suggestions is dynamically adjusted based on the level of ambiguity in the user's question. Additionally, Odin can learn user's preferences for specific schema components through their feedback, enhancing future recommendations.

Internally, Odin operates using a Generate-Select paradigm. Given a semantically ambiguous natural language question, the **Generator** is tasked with producing the set of all potentially correct SQL queries. Traditional approaches, such as beam search or diversity-promoting techniques like nucleus sampling [9], often fail to generate queries that reflect schema ambiguity, as noted in [2]. To overcome this limitation, Odin uses an iterative strategy: the Generator sequentially produces candidate SQL queries by selectively modifying the information provided to the LLM. Specifically, it removes certain schema elements from previously generated SQL queries, encouraging the LLM to explore different schema components and generate diverse queries.

The generator may produce incorrect SQL queries by omitting too many important schema components or by generating results that do not align with user preferences. The **Selector** component addresses this by filtering out the inaccuracies to reduce the set size while ensuring the correct SQL query is retained, thus maintaining high recall. This process is framed within the conformal prediction framework [24], which is commonly used to provide confidence in the outputs of machine learning models in critical fields such as medical diagnosis [28]. Conformal prediction provides concrete guarantees on recall, ensuring that it does not fall below a specified threshold.

After the selection phase, users can choose their preferred SQL query from the remaining options. Odin learns user preferences from this feedback, allowing it to refine its recommendations and better align future outputs with the user's specific preferences.

Our evaluation demonstrates that the set of SQL queries recommended by Odin contains the correct query 1.5–2× more often, while maintaining a result set that is 2–2.5× smaller, compared to other baselines on the AmbiQT benchmark, which contains various forms of schema ambiguity[1].

In summary, we make the following contributions:

- We present Odin, a system designed to handle and resolve ambiguity in NL2SQL tasks (Section 4).
- We introduce a novel SQL generation-selection strategy that outperforms LLM-based sampling (Sections 5 and 6).
- We offer a personalization algorithm for Odin to learn user preferences (Section 7).
- We evaluate Odin against state-of-the-art baselines and demonstrate its effectiveness (Section 8).

2 RELATED WORK

NL2SQL. Generating accurate SQL queries from natural language questions (NL2SQL) is a long-standing challenge due to the complexities in user question understanding, database schema comprehension, and SQL generation [11, 15, 21]. Recently, large language models (LLMs) have demonstrated significant capabilities in natural language understanding as the model scale increases. Many LLM-based NL2SQL solutions [7, 15, 17, 26, 31] have emerged.

[1]We use a modified version of this benchmark to introduce additional ambiguity in the questions.

SC-Prompt [7] divides the NL2SQL task into two simpler sub-tasks (i.e., a structure stage and a content stage). Structure and content prompt construction and fine-grained constrained decoding are proposed to tackle these two sub-tasks, respectively. CodeS [17] adopts an incremental pre-training approach to enhance the SQL generation capability. It further addresses the challenges of schema linking and domain adaptation through prompt construction and data augmentation. MAC-SQL [31] is an LLM-based multi-agent collaborative framework, in which a decomposer agent leverages few-shot chain-of-thought reasoning for SQL generation and two auxiliary agents utilize external tools or models to acquire smaller sub-databases and refine erroneous SQL queries. CHESS [26] introduces a new pipeline that consists of effective relevant data/context retrieval, schema selection, and SQL synthesis. Also, CHESS is equipped with an adaptive schema pruning technique based on the complexity of the problem and the model's context size.

These methods focus on generating more accurate SQL queries for a given NL question. However, they neither explicitly handle schema ambiguity nor learn the preferences from users. Consequently, these state-of-the-art methods are not able to learn from user feedback and improve their SQL generation progressively.

Ambiguity in SQL. While ambiguity has been extensively studied in many fields of NLP [4, 20], it has not been explored much in NL2SQL. Hou et al., [10] introduce an uncertainty decomposition framework for LLMs in general, which can be applied to any pre-trained LLM. CAmbigNQ [14] focuses on ambiguous questions in open-domain question answering, tackling the problem in three steps, ambiguity detection, clarification question generation, and clarification-based QA. CLAM [13] is a framework that drives language models to selectively ask for clarification about ambiguous user questions and give a final answer after receiving clarification for open-domain QA. These works focus on clarifying the intent of the question, rather than resolving ambiguities within the data where the answer resides.

In NL2SQL, Wang et al. [30] tackle ambiguity in SQL arising from related column names. The proposed method relies on labeling words of the text and does not generalize to other types of ambiguity beyond column ambiguity. AmbiQT [2] represents the first open benchmark for testing coverage of ambiguous SQLs. It introduces a decoding algorithm that searches the SQL logic space by combining plan-based template generation with a beam-search-based infilling. However, it does not capture and adapt to user preferences over the course of multiple questions.

3 PROBLEM FORMULATION

Traditional NL2SQL systems typically generate a single SQL query in response to a user's question. The main goal is for the generated SQL query to match the ground truth SQL query in terms of execution result. However, this approach may fall short in cases where the question contains ambiguity. In such instances, the generated query might not align with what the user intended. To address this, we propose developing an NL2SQL system that generates a set of possible queries, allowing the user to select the one that best meets their needs.

Given a user question Q, the system generates a set of SQL queries to potentially answer the question. The process can be expressed as:

$$\{\text{SQL}_1,\ldots,\text{SQL}_k\} = NL2SQL(Q)$$

We assume that although the natural language question input to the system contains semantic ambiguities, there is nonetheless a single "correct" SQL query which captures the user's true intent and preferences, which we refer to as the ground truth query SQL_{GT}. The system aims to maximize the likelihood that the execution of any SQL query in the final set returns the same result as the execution of SQL_{GT}. Let $EX(SQL)$ denote the execution result of query SQL. Let $A(Q)$ denote the execution accuracy for question Q, defined as:

$$A(Q) = \begin{cases} 1 & \text{if } \exists SQL \in \text{NL2SQL(Q) such that } EX(SQL) = EX(SQL_{GT}) \\ 0 & \text{otherwise} \end{cases}$$

The average accuracy over a workload of natural language questions $W = \{Q_1, Q_2, \ldots, Q_N\}$ is:

$$\text{AvgAcc}(W) = \frac{1}{N}\sum_{i=1}^{N} A(Q_i)$$

In concept, generating all possible SQL queries for a question will achieve 100% accuracy, but it is impractical as it overwhelms users with too many options. Maximizing accuracy alone does not guarantee a good user experience. When users are presented with an excessive number of choices, identifying the correct query becomes challenging. Therefore, it is essential to balance accuracy and the size of the result set to enhance user experience.

The optimization goal of the system is to maximize the likelihood of including the correct SQL query in a manageable set of results (K), ensuring efficiency and usability. The average number of SQL queries shown over the workload W is:

$$\text{AvgResultSize}(W) = \frac{1}{N}\sum_{i=1}^{N} |NL2SQL(Q)|$$

The overall objective is to maximize average accuracy while limiting the average number of results.

$$\textbf{max}\text{AvgAcc}(W), \ \textbf{subject to} \ \text{AvgResultSize}(W) \le K$$

This balance is key for optimizing both system performance and user experience.

4 ODIN SYSTEM OVERVIEW

Odin is an NL2SQL recommendation engine designed to assist users in managing schema ambiguities within their databases by generating multiple SQL query options and further resolving these ambiguities through learning user preferences. Upon receiving a natural language user question, Odin generates several potential SQL queries, as demonstrated in Fig. 1(A). Odin personalizes query generation by incorporating user feedback. The core novelty of Odin compared to other NL2SQL systems lies in its ability to generate a small, accurate set of SQL queries tailored to user preferences. Internally, Odin is composed of three key components: the Generator, the Selector, and the Personalizer.

Generator: This component generates potential SQL candidates based on the user's question. A typical approach used by other NL2SQL systems [27, 32] for generating multiple SQL query candidates is to repeatedly call the LLM with the same prompt, using high temperatures and sampling techniques such as nucleus sampling [9] to induce different output SQL queries with each call. However, [2] shows that this approach does not produce sufficiently diverse SQL queries, because LLMs tend to produce simple variants of the same SQL query during sampling. For instance, the model might generate queries such as `select * from students`, `SELECT * FROM students`, or `select * from students;` with only minor cosmetic variations.

Odin's Generator improves on this baseline approach by incorporating previously-generated queries into the generation process. Specifically, our method selectively masks certain schema elements used in the previously-generated SQL queries, encouraging the LLM to explore alternative schema elements. For instance, as illustrated in Figure 1(B), the initial SQL query is generated using the entire schema. Key columns, such as `birthplace` and `roll_num`, are then marked as candidates for masking. By excluding these columns in subsequent LLM calls, new *masked schemas* are generated, leading to different SQL queries. For example, in the newly generated SQL query, the `origin` column replaces `birthplace`, and `id` replaces `roll_num`. Although this approach can produce diverse queries, it risks inefficiency due to the exponential number of possible masked schemas. This is particularly problematic when LLM calls are computationally expensive and should be minimized. Section 5 describes our approach to generate diverse SQL queries while minimizing the number of LLM calls.

Selector: The generation algorithm may occasionally mask schema elements for which no reasonable substitutes exist, leading the LLM to produce incorrect SQL queries. Additionally, some generated queries might not align with user preferences. The primary objective of Odin is to maintain a compact set of generated SQL queries while ensuring accuracy. Removing these incorrect and misaligned queries can help Odin reduce the set size.

To address this issue, after Odin's Generator has produced a set of candidate SQL queries, Odin's Selector filters out candidate SQL queries which are likely to be incorrect, ensuring that nearly all correct SQL queries are preserved. To ensure accuracy in this filtering process, we employ the conformal prediction framework [24], which provides statistical guarantees for our ability to retaining correct queries. Specifically, we evaluate candidate SQL queries with a scoring function, selecting those exceeding a threshold (e.g., Fig. 1(C) shows an example where candidates with scores above 0.8 are retained). The effectiveness of this process hinges on the scoring function and threshold selection, which we discuss further in Section 6.

Personalizer: Users may prefer that SQL queries use certain tables and columns over others, and learning these preferences can enhance recommendation quality. Since preferences are specific to the application, schema, or database, they require user input. These preferences must also be conveyed to the Generator and Selector components to personalize outputs.

After Odin produces the set of SQL queries for a given user question, the user is asked to select the correct SQL query from the output set, if any. This feedback is transformed into textual hints, which the Generator and Selector use for personalization. For instance (Fig. 1(D)), if a user selects a SQL query that uses the `origin` column instead of `birthplace`, the system infers that the term *hometown* from the user's query maps to `origin` rather than to `birthplace`. The hint generator then processes this feedback, along with the user's current question, to perform schema linking (i.e., mapping specific entities from the user's input question to corresponding

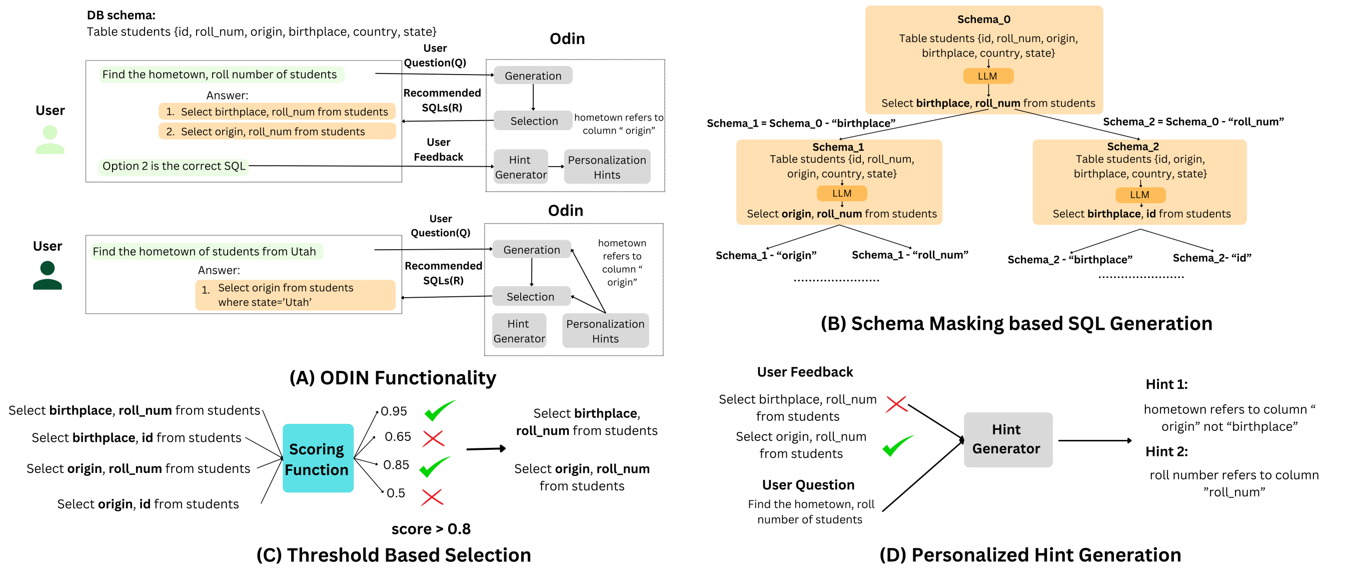

DB schema:
Table students {id, roll_num, origin, birthplace, country, state}
User
Find the hometown, roll number of students
Answer:
1. Select birthplace, roll_num from students
2. Select origin, roll_num from students
Option 2 is the correct SQL
User Question(Q)
Recommended SQLs(R)
User Feedback
Odin
Generation
Selection
Hint Generator
Personalization Hints
hometown refers to column " origin"
User
Find the hometown of students from Utah
Answer:
1. Select origin from students where state='Utah'
(A) ODIN Functionality
Schema_0
Table students {id, roll_num, origin, birthplace, country, state}
LLM
Select birthplace, roll_num from students
Schema_1 = Schema_0 - "birthplace"
Schema_2 = Schema_0 - "roll_num"
Schema_1
Table students {id, roll_num, origin, country, state}
Select origin, roll_num from students
Schema_2
Table students {id, origin, birthplace, country, state}
Select birthplace, id from students
Schema_1 - "origin"
Schema_1 - "roll_num"
Schema_2 - "birthplace"
Schema_2- "id"
(B) Schema Masking based SQL Generation
Select birthplace, roll_num from students
Select birthplace, id from students
Select origin, roll_num from students
Select origin, id from students
Scoring Function
0.95
0.65
0.85
0.5
score > 0.8
(C) Threshold Based Selection
User Feedback
Select birthplace, roll_num from students
Select origin, roll_num from students
User Question
Find the hometown, roll number of students
Hint Generator
Hint 1:
hometown refers to column " origin" not "birthplace"
Hint 2:
roll number refers to column "roll_num"
(D) Personalized Hint Generation

Figure 1: Odin Overview

schema elements). Through this schema linking, Odin can map natural language phrases like *hometown* and *roll number* from the user's question to the `origin` and `roll_num` columns, respectively. These personalized hints are stored and later used by the Generator and Selector to align future SQL queries with user preferences. In Fig. 1(A), when the user asks, "*hometown of students from Utah*", the system uses the hint that *hometown* refers to `origin`, enabling it to generate the correct SQL query. We describe the personalization process in further detail in Section 7.

5 GENERATOR

For a given question that can be answered with multiple possible schema components, Odin ensures all potential SQL queries using these components are generated and presented to the user. The generator in Odin handles this by producing the set of potential queries. A common method in LLM pipelines for generating diverse answers is using high-temperature sampling, where the temperature controls randomness. Higher temperatures lead to more varied and creative responses, while lower temperatures produce more focused, predictable results.

One might assume that high temperatures would generate diverse SQL queries, but in practice, it leads to only superficial differences, such as varying table/column aliases, using equivalent functions, or altering the SQL structure with subqueries. While these changes affect the query's appearance, they don't impact execution, resulting in only cosmetic diversity in generated SQL.

For example, consider the question: *"List the names of customers who have placed orders over $1,000 in the past six months."* At a high temperature setting, the model might generate the following SQL in one run:

```
SELECT Name FROM Customers
JOIN Orders ON Customers.CustomerID = Orders.CustomerID
WHERE Orders.TotalAmount > 1000
AND Orders.OrderDate >= DATE_SUB(CURDATE(), INTERVAL 6 MONTH);
```

This SQL joins the `Customers` and `Orders` tables and filters for orders over $1,000 within the last six months. On a second run, it might produce:

```
SELECT c.Name FROM Customers c
INNER JOIN Orders o ON c.ID = o.CustomerID
WHERE o.TotalAmount > 1000
AND o.OrderDate >= DATE_SUB(CURDATE(), INTERVAL 6 MONTH);
```

This version simply uses different aliases for the tables compared to the first query, but it will yield the same execution result. Finally, on a third run, the model might generate:

```
SELECT Name FROM Customers
WHERE CustomerID IN (
    SELECT CustomerID FROM Orders
    WHERE TotalAmount > 1000
  AND OrderDate >= DATE_ADD(CURRENT_DATE, INTERVAL -6 MONTH)
);
```

Although this query uses a subquery and a different date function, it yields the same output as the previous two queries. This example demonstrates how high temperatures in LLMs generate superficial variations in NL2SQL tasks without changing the effective SQL, as studied in [2].

Since high-temperature sampling produces only superficial diversity in results, we can instead prompt the LLM to generate SQL queries with different execution outcomes. To achieve this, we provide the LLM with all previously generated SQL queries and explicitly instruct it (via the LLM prompt) to create a new SQL query that yields a distinct execution result. We refer to this approach as *ForcedDiversity*. Our empirical results show that *ForcedDiversity* generates more truly diverse queries. However, it starts to repeat outcomes when too many SQL queries are included in the prompt.

To improve upon previous methods, Odin leverages the structure of the NL2SQL task to enhance diversity by introducing controlled perturbation of schema information in the prompt. By masking certain parts of the schema used in earlier queries, the LLM is forced to rely on different schema elements, leading to more varied SQL queries. For example, for the question *"Find the hometown of students"*, if the database has both `birthplace` and `origin` columns, and the LLM initially uses `birthplace`, we can remove knowledge of this column's existence in subsequent prompts to encourage the LLM to generate a query using `origin`, leading to more diverse SQL queries.

We use the term *masked schema* to refer to a subset of the full schema, i.e., a schema in which certain elements have been masked. We first present a simple algorithm for schema masking. We then use the limitations of this simple algorithm to motivate the schema masking algorithm used by Odin's Generator.

5.1 Naive Schema Masking Algorithm

A naive version of the schema masking algorithm is shown in Fig. 1(B). It generates SQL queries by progressively restricting a given schema and exploring different masked schemas in a tree-like structure. The root node of the tree structure represents the complete schema of the database. Each node in this tree represents a masked schema that is a subset of the complete schema. At each node, a SQL query is generated based on the current schema, which is then added to the set of SQL queries. The algorithm then modifies the current schema of that tree node by removing individual columns used in the generated query, which produces new schemas to explore in child nodes. This process continues until all relevant masked schemas are exhausted. In Fig. 1(B), the root node represents the complete schema, and an initial SQL query is generated from it. Columns such as `birthplace` and `roll_num`, which are used in this query, are then removed, leading to modified schemas that are used to generate additional SQL queries. This approach ensures that the SQL queries generated by the child nodes are different from those of their parent nodes. This is straightforward to verify because the schemas used to generate child queries are missing at least one column present in the parent SQL query. However, the naive schema masking algorithm has two main drawbacks.

First, this algorithm might result in redundant explorations of the same masked schema. The algorithm does not ensure that descendants of a node explore distinct schemas. In some scenarios, the exploration paths of different branches may overlap. For example, in Fig. 1(B), child schemas of the root node are generated by removing columns `birthplace` and `roll_num`. The descendants of these child schemas might explore the same schemas in different orders (e.g., removing `birthplace` then `roll_num`, vs. removing `roll_num` then `birthplace`), leading to duplicated effort.

Second, the algorithm operates without accounting for practical resource constraints, such a constraint on the number of calls to an LLM. In real-world applications, the number of LLM calls is limited, but the exploration space of this algorithm grows exponentially with the size of the schema, leading to inefficient use of resources.

5.2 Odin's Schema Masking Algorithm

Odin's schema masking algorithm introduces two main improvements over the naive algorithm presented above: duplicate schema detection and a greedy exploration strategy for the search tree of schemas. The key idea behind duplicate schema detection is to maintain a record of schemas seen so far and to not explore redundant nodes. To deal with the issue of limited resources, we introduce a greedy tree search strategy that uses a priority queue to explore nodes based on a scoring system. These scores reflect the relevance of schema of the current node to the entities mentioned in the user's question, allowing the algorithm to focus on the most promising exploration paths while staying within resource constraints.

Algorithm 1 shows the pseudocode for the algorithm. The algorithm takes the user question, database schema and a limit of LLM calls as input (Line 12). If the LLM fails to find relevant entities in the masked schema and is unable to generate a valid SQL query, the exploration may terminate early. The limit on LLM calls represents only the maximum number of attempts, not a guarantee that all calls will be used. To prioritize exploration, the algorithm maintains a priority queue of nodes (i.e., schemas) to explore, where each node's priority is determined by its relevance score, computed by the `Cal_Score` function (see Section 5.3). At each iteration, the node with the highest score is selected for exploration (Line 19).

When a node is explored, its corresponding schema is used to generate a SQL query, which is added to the results (Lines 20-22). Similar to the naive algorithm, new potential nodes (i.e., schemas) are generated by removing columns used in the SQL query (Lines 23-29). These new schemas are then scored using a relevance function. If the new schema has already been explored or is empty, it is discarded; otherwise, it is added to the priority queue. This ensures that the exploration is both resource-efficient and free from duplicate schemas. In summary, the algorithm combines a greedy budget-conscious exploration strategy with a simple mechanism for avoiding redundant searches.

5.3 Scoring Masked Schemas

The schema scoring function, `Cal_Score`, is crucial to the generation algorithm, as a noisy scoring mechanism can hinder efficient exploration. A masked schema's score intuitively reflects the likelihood that a plausibly-correct SQL query for the user's question can be formed from the elements of the masked schema. An *entity* refers to information from the user's question, such as *hometown* or *roll number*. Each entity in the user's question should be well-represented by the masked schema in order to produce a SQL query. If a schema poorly represents any entity in the user's question, then the schema should intuitively get a lower score.

For example, consider the scenario in Fig. 1(B). The entity *hometown* could plausibly be represented by either the `birthplace` column or the `origin` column. In contrast, the entity *roll number* can only be correctly represented by the column `roll_num`. This distinction suggests that removing `birthplace` from the schema is less impactful than removing `roll_num`. The `Cal_Score` function reflects this intuition by computing the similarity between each entity in the question to its best-matching column in the schema and using the minimum similarity score across all entities as the schema's score.

In Algorithm 2, the function starts by extracting entities from the query (Line 10). It then calculates the maximum similarity score for each entity across all schema columns, storing these scores (Lines 11-18). The final schema score is determined by the lowest score among all entities, highlighting the schema's weakest representation (Line 19). The `Extract_Entities` function identifies entities in the user's question with an LLM call. The `Cal_Sim` function, which measures entity-column similarity, utilizes SBERT similarity scores [22].

Algorithm 1 Greedy Tree Search with Resource Constraints

```
Input:
    $f\_sch$ - Full schema
    $q$ - Initial query
    $max\_calls$ - Maximum number of LLM calls
Output:
    $final\_queries$ - List of SQL queries
Algorithm:
    $NL2SQL$ - Generates SQL for a given schema and user question
    $Col\_Used$ - Returns columns used in a SQL query
    $Remove\_Col$ - Removes a column from the schema
    $Cal\_Score$ - Calculates relevance score for a schema
function GenSQLQueries($f\_sch, q, max\_calls$)
    $final\_queries \leftarrow []$
    $queue \leftarrow$ PriorityQueue()
    $schema\_seen \leftarrow []$
    $queue$.push(($f\_sch$,1.0))                ▷ Initial score is 1
    $llm\_calls \leftarrow 0$
    while $queue$ is not empty and $llm\_calls < max\_calls$ do
        ($curr\_sch$,_) $\leftarrow queue$.pop()
        $sql \leftarrow NL2SQL(curr\_sch, q)$
        $llm\_calls \leftarrow llm\_calls + 1$
        $final\_queries$.append($sql$)
        for each $col$ in Col_Used(sql) do
            $new\_sch \leftarrow Remove\_Col(curr\_sch, col)$
            $score \leftarrow Cal\_Score(new\_sch, query)$
            if $new\_sch$ is not empty then
                if $new\_sch$ not in $schema\_seen$ then
                    $queue$.push(($new\_sch, score$))
                    $schema\_seen.add(new\_sch)$
    return $final\_queries$
```

Algorithm 2 Calculate Schema Relevance Score

```
Input:
    $schema$ - Current schema
    $query$ - User question
Output:
    $score$ - Relevance score of the schema
Helper Functions:
    $Extract\_Entities$ - extracts entities from the user question
    $Cal\_Sim$ - gives similarity score between an entity and a column
function Cal_Score($schema, question$)
    $entities \leftarrow$ Extract_Entities($question$)
    $entity\_scores \leftarrow []$
    for each $entity$ in $entities$ do
        $max\_similarity \leftarrow -\infty$
        for each $col$ in $schema$ do
            $similarity \leftarrow$ Cal_Sim($entity, col$)
            if $similarity > max\_similarity$ then
                $max\_similarity \leftarrow similarity$
        $entity\_scores$.append($max\_similarity$)
    $score \leftarrow$ min($entity\_scores$)
    return $score$
```

Note that when computing the similarity between an entity and a column, considering the table leads to better semantic results. For example, replacing the `address` column from the `student` table with the `address` column from the `student_registration` table may be more relevant than using the `address` column from the `teachers` table. Thus, we include table semantics when computing similarity as well.

6 SELECTOR

The Generator may be overeager when masking schema elements, leaving no suitable schema elements for a given question entity, which can lead the LLM to select incorrect tables or columns or omit necessary elements, resulting in an inaccurate SQL query. Additionally, if user preferences for certain tables or columns are masked, the generated query may not align with their intent. As a result, queries produced by the Generator may be incorrect or misaligned. The Selector's primary role is to eliminate flawed SQL queries while ensuring the correct one is retained. We can express this objective as follows:

$$\begin{aligned} \textbf{minimize} \quad & |\mathrm{Sel}(\mathrm{Gen}(Q))| \\ \textbf{subject to} \quad & \Pr\big(\mathrm{EXM}(\mathrm{Sel}(\mathrm{Gen}(Q)),\mathrm{SQL}^{gt})=1 \mid \\ & \quad \mathrm{EXM}(\mathrm{Gen}(Q)),\mathrm{SQL}^{gt}=1\big) > (1-\alpha) \end{aligned} \tag{1}$$

In this formulation, $\mathrm{EXM}(S,\mathrm{SQL}^{gt})$ checks if any query in S produces the same execution result as the ground truth query SQL^{gt}. $\mathrm{Gen}(Q)$ is the set of SQL queries produced by the Generator for a query Q, and $\mathrm{Sel}(\mathrm{Gen}(Q))$ is the filtered subset of queries produced by the Selector. The parameter α, typically between 1% and 5%, represents the acceptable margin of error. This ensures that the selected set retains at least one query matching the ground truth, with the probability exceeding $1-\alpha$. The constraint minimizes the risk of discarding the correct query while reducing the number of SQL queries return to the user, balancing accuracy and efficiency.

The objective of the Selector closely aligns with the conformal prediction framework [1, 24], which is commonly used in classification tasks. In conformal prediction, the goal is to select a subset of labels such that the true label is included with high probability. The key idea behind conformal prediction is straightforward: a scoring function is used to assess how likely a label is incorrect, and all labels below a carefully chosen threshold are selected. This threshold is determined using a calibration set, ensuring that the high-probability guarantees hold, assuming new queries come from the same distribution as the calibration set. Similarly, we aim to select a subset of SQL queries such that the correct one is retained with high probability. We provide an overview of conformal prediction in Section 6.1, explain how we map our problem to this framework in Section 6.2, and describe our scoring function in Section 6.3.

6.1 Background on Conformal Prediction

Originally proposed by [29], conformal prediction provides a way to turn heuristic notions of uncertainty from machine learning models into rigorous sets that are guaranteed to contain the true outcome with a specified probability. This framework can be applied to both regression and classification tasks, making it highly adaptable for various machine learning applications. The central idea of conformal prediction is to construct a set of possible outcomes for a new input

X_{test} such that the set will contain the true outcome Y_{test} with a user-specified confidence level $1-\alpha$. The procedure for generating these prediction sets is non-parametric and relies on a calibration dataset to quantify the uncertainty of the model's predictions.

The conformal prediction framework proceeds through the following key steps:

1. Heuristic Uncertainty Estimate: Start with a pre-trained model that provides a score function $s(x,y) \in \mathbb{R}$ for each input-output pair (x,y). This score function reflects the level of disagreement between the input x and the predicted or true output y. A higher score indicates worse agreement between x and y, i.e., more uncertainty.

2. Calibration: To quantify uncertainty, the model is first calibrated using a set of observed data points $\{(X_1,Y_1),...,(X_n,Y_n)\}$. For each pair (X_i,Y_i), the score $s(X_i,Y_i)$ is computed, yielding a set of calibration scores $s_1 = s(X_1,Y_1),...,s_n = s(X_n,Y_n)$. A quantile threshold $\hat{s}$ is then selected such that

$$\hat{s} = \left(\frac{\lceil (1+n)(1-\alpha) \rceil}{n}\right)\text{-th quantile of } \{s_1,...,s_n\}. \tag{2}$$

This quantile ensures that future prediction sets will contain the true outcome with probability at least $1-\alpha$.

3. Prediction Set Formation: For a new input X_{test}, the score function is evaluated for various possible outputs y. The prediction set $C(X_{\text{test}})$ is then formed by including all outputs y such that the score function satisfies:

$$C(X_{\text{test}}) = \{y : s(X_{\text{test}},y) \leq \hat{s}\}.$$

This ensures that the prediction set contains all outcomes where the model's uncertainty (as captured by the score function) is below the threshold $\hat{s}$.

Theorem 6.1 (Conformal Prediction Guarantee, [29]). *Given a calibration dataset $\{(X_1,Y_1),...,(X_n,Y_n)\}$ drawn i.i.d. from the same distribution as the test point (X_{test}, Y_{test}), conformal prediction constructs a prediction set $C(X_{test})$ using the threshold in Eq. (2) for the true outcome Y_{test}. The prediction set satisfies the following coverage guarantee:*

$$P(Y_{test} \in C(X_{test})) \geq 1-\alpha,$$

where α is a user-specified significance level. This guarantee holds regardless of the underlying distribution of the data.

6.2 Mapping Selector to Conformal Prediction

In this subsection, we demonstrate how the Selector aligns with the conformal prediction framework. By establishing this connection, we leverage the statistical guarantees provided by conformal prediction to ensure that the correct SQL query is retained with high probability after the selection process. Conformal prediction constructs prediction sets that contain the true outcome with a specified confidence level $1-\alpha$. To map our problem to this framework, we define the following components:

Inputs and Outputs: In our setting, the input X is the user's natural language query Q, and the output Y is a candidate SQL query q generated by the language model (LLM).

Score Function: We define a score function $s(Q,q)$ that quantifies the likelihood of the SQL query q being *incorrect* for the given input Q. A higher score indicates a higher chance that q is incorrect. The specific design of this score function is detailed in Section 6.3.

Calibration Set: The calibration set consists of pairs $\{(Q_i,\text{Gen}(Q_i))\}$, where Q_i are natural language queries and $\text{Gen}(Q_i)$ are the corresponding sets of SQL queries generated by Odin . Importantly, we only look at pairs for which a query with the same execution result to that of the correct SQL query SQL_i^{gt} is present in $\text{Gen}(Q_i)$. For each pair, we compute the score $s(Q_i,q)$ for all queries $q \in \text{Gen}(Q_i)$, focusing on the score distribution for correct SQL queries.

Quantile Threshold: We determine the threshold $\hat{s}$ by analyzing the scores of correct SQL queries in the calibration set, specifically using the quantile defined in Eq. (2).

Prediction Set Formation: For a new input Q_{test}, we generate a set of candidate SQL queries $\text{Gen}(Q_{\text{test}})$. We compute the scores $s(Q_{\text{test}},q)$ for each $q \in \text{Gen}(Q_{\text{test}})$. The selector then forms the prediction set $\text{Sel}(\text{Gen}(Q_{\text{test}}))$ by including all queries with scores less than or equal to $\hat{s}$:

$$\text{Sel}(\text{Gen}(Q_{\text{test}})) = \{q \in \text{Gen}(Q_{\text{test}}) \mid s(Q_{\text{test}},q) \leq \hat{s}\}.$$

Mapping the selector to the conformal prediction framework provides the coverage guarantee from Eq. (1). The effectiveness depends on the quality of score function $s(Q,q)$ (see Section 6.3 for details).

6.3 Scoring Functions

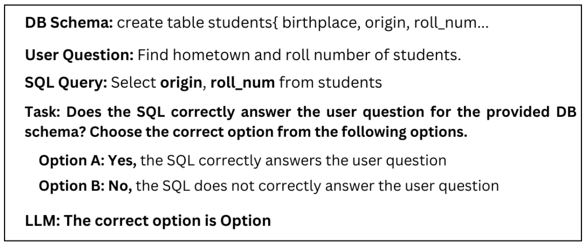
In the Selector, the scoring function evaluates each SQL query generated by the model. The goal is to assign *low scores* to correct SQL queries and *high scores* to incorrect ones. Ideally, the ground truth SQL query would get a score of zero, while all others get higher scores, allowing for confident elimination of incorrect queries. However, designing a perfect scoring function is difficult due to the inherent complexity of natural language and SQL semantics.

The scoring function should capture the *semantic alignment* between the user's question and the SQL query, focusing on the relevant tables and columns. To achieve this, we leverage the capabilities of language models to understand and represent semantic relationships. We develop two scoring functions: an *LLM-based function* that directly evaluates the SQL query using an LLM, and an *SBERT-based function* that heuristically decomposes the task and uses Sentence-BERT (SBERT) [22] to compute semantic similarities. Each scoring function is described below.

6.3.1 LLM-based Scoring Function. To develop the LLM-based scoring function, we utilize the language model's ability to understand complex instructions by prompting it to evaluate whether a given SQL query correctly answers the user's question. One method is to ask the LLM to assign a score or probability directly [35], while another is to use the logit probabilities of specific tokens. We adopt the logit probabilities approach for finer-grained scoring.

We use a prompt that asks the LLM if the SQL query answers the question, providing two options: *A.* Yes, and *B.* No (as shown in Fig.2). We use the logit probability of option B as the score. A lower probability for B indicates a higher likelihood that the SQL query is correct, aligning with our goal of assigning lower scores to correct queries.

6.3.2 SBERT-based Scoring Function. Although LLMs perform well, they can be resource-intensive. As a more efficient alternative, we propose an SBERT-based scoring function. SBERT computes semantic similarities between sentences or phrases [22], which we leverage by breaking down the SQL query scoring task into smaller sub-tasks involving entity similarity computations.



DB Schema: create table students{ birthplace, origin, roll_num...

User Question: Find hometown and roll number of students.

SQL Query: Select **origin**, **roll_num** from students

Task: Does the SQL correctly answer the user question for the provided DB schema? Choose the correct option from the following options.

Option A: Yes, the SQL correctly answers the user question

Option B: No, the SQL does not correctly answer the user question

LLM: The correct option is Option

Figure 2: Prompt template used to score SQL queries using LLMs

Our approach is based on the observation that a correct SQL query should represent all entities mentioned in the user's question via its tables and columns, similar to the schema scoring function used in generation. The SBERT-based scoring follows Algorithm 2, but focuses on the columns in the SQL query rather than the masked schema. By negating this score, we assign lower scores to queries that better represent the user's question. Queries missing entities receive higher scores, allowing the selector to filter them out. This method efficiently captures semantic alignment and is suitable for resource-limited scenarios.

7 PERSONALIZATION

As mentioned earlier, ambiguity in NL2SQL often arises when a database contains multiple similarly named tables or columns. For example, if a user asks, *What were the total sales last year?*, but the database includes both `gross_sales` and `net_sales`, it is unclear which column the user intends to reference. User preferences are reflected in their choice of specific schema components. One user might consistently prefer the `gross_sales` column when referring to total sales, while another may focus on `net_sales`. These preferences are often tied to how the user phrases their query. For instance, one user might use the term *total sales* to mean `gross_sales`, while another might mean `net_sales` when referring to *final sales*. It is crucial to map the user's phrases or entities to the corresponding schema components.

Capturing user preferences can improve future recommendation, like in the previous example, if we know that a user associates `gross_sales` with *total sales*, future queries can avoid incorrectly selecting the `net_sales` column in similar situations. While user preferences improve recommendation quality, the challenge lies in capturing them effectively. Without explicit feedback, it is difficult for the system to infer such preferences. Moreover, even after preferences are captured, integrating them into future query generation is not straightforward, as it requires biasing the system towards them. The Personalizer component addresses these challenges.

First, users select the correct SQL query from a set of displayed options, and this feedback forms the foundation for personalization in Odin. This feedback is then transformed into a format that influences both the Generator and Selector components of the system. Specifically, we provide these preferences in a textual format that biases the LLM's output. The key information conveyed in the textual hints is that when a user refers to a particular entity, a specific schema component should be chosen over alternatives. Odin generates these hints by mapping the entities mentioned in the user's question to the corresponding schema components in the selected SQL query. Through these techniques, Odin learns to associate entities in user questions with selected schema components, ensuring future queries are personalized based on past feedback.

7.1 Generating Textual Hints

Textual hints are used to guide both the Generator and Selector towards user preferences. The key information that textual hints capture is that when a user references an entity in their question, they prefer a specific schema components. We frame this as learning the correct schema linking based on the user's question and their preferred SQL query. Given a question Q, the correct SQL query (SQL_{True}), and a set of incorrect queries (SQL_1,...,SQL_T), the task is to map entities E_1,...,E_P in the question to schema components in SQL_{True} and the incorrect queries. Incorrect mappings help reduce the importance of wrong components, guiding the system towards accurate ones.

For each entity E_i, Odin learns its mapping to the schema component in SQL_{True}. For instance, if E_1 refers to *total sales*, and SQL_{True} maps it to the `gross_sales` column from the `customer_sales` table, while incorrect queries map it to `net_sales`, Odin generates the following textual hint: *When referring to **total sales**, the user prefers the **customer_sales.gross_sales** column over **customer_sales.net_sales**.*

The key subroutine for generating textual hints is schema linking, which maps entities from the user's question to the relevant tables and columns. Schema linking is a well-studied problem with approaches using general LLMs [8], specialized LLMs [18], and SBERT models [3, 33]. While any method could be used, we propose a heuristic SBERT-based algorithm for greater efficiency.

The algorithm for generating textual hints, as detailed in Algorithm 3, links entities from the user's question to schema components and creates textual hints summarizing the correct mappings. This process begins with three inputs: the user's question, the correct SQL query, and a set of incorrect SQL queries. The output is a list of textual hints based on the correct schema mappings.

The core function, `GenerateHints`, starts by extracting entities from the user's question using the `Extract_Entities` function. For each entity, the `SchemaMap` function determines the most likely schema component in the correct SQL query. Simultaneously, the algorithm gathers mappings from incorrect SQL queries and compares them with the correct mappings, recording any discrepancies as incorrect mappings. The `SchemaMap` function computes the similarity between an entity and each column in the SQL query using `Cal_Sim`, selecting the column with the highest similarity score as the correct mapping. This ensures that each entity is accurately linked to the most relevant schema component, enhancing the precision of the generated hints.

Finally, the `Format_Hint` function integrates the entity, the correct schema mapping, and the list of incorrect mappings into a template for generating hints. After processing all entities, the function returns the list of textual hints.

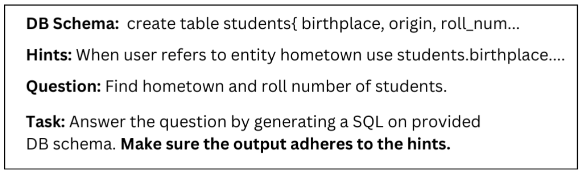
7.2 Using Personalization Hints

We leverage LLMs' in-context learning to incorporate personalization hints during both the Generator and Selector stages. In the Generator stage, hints guide SQL query generation by including

Algorithm 3 Schema Linking and Hint Generation

```
Input:
    $Q$ - User question
    $SQL_{\text{True}}$ - Correct SQL query
    $SQLs_{\text{Incorrect}}$ - List of incorrect SQL queries
Output:
    $hints$ - List of textual hints for each entity
Helper Functions:
    $Extract\_Entities$ - extracts entities from the user question
    $Cal\_Sim$ - gives similarity score between an entity and a column
    $Format\_Hint$ - generates textual hint using a template and provided entity ans schema components.
function GenerateHints($Q$, $SQL_{\text{True}}$, $SQLs_{\text{Incorrect}}$)
    $entities \leftarrow$ Extract_Entities($Q$)
    $hints \leftarrow [\,]$
    for each $entity$ in $entities$ do
        $correct\_map \leftarrow$ SchemaMap($entity$,$SQL_{\text{True}}$)
        $incorrect\_map \leftarrow [\,]$
        for each $SQL$ in $SQLs_{\text{Incorrect}}$ do
            $mapping \leftarrow$ SchemaMap($entity$,$SQL$)
            if $mapping \neq correct\_map$ then
                $incorrect\_map$.append($mapping$)
        $hint \leftarrow$ FormatHint($entity$,$correct\_map$,$incorrect\_map$)
        $hints$.append($hint$)
    return $hints$
function SchemaMap($entity$, $SQL$)
    $max\_sim \leftarrow -\infty$
    $best\_mapping \leftarrow None$
    for each $col$ in $SQL$ do
        $sim \leftarrow$ Cal_Sim($entity$,$col$)
        if $sim > max\_sim$ then
            $max\_sim \leftarrow sim$
            $best\_mapping \leftarrow col$
    return $best\_mapping$
```

them in the LLM's context, as shown in Figure 3. For the Selector, if using an LLM-based scoring function, hints are similarly added to the LLM's context. However, with an SBERT-based scoring function, integrating textual hints is less straightforward. To address this, we fine-tune the SBERT model by adjusting its representations, ensuring that preferred schema components align closely with the corresponding entities, while non-preferred components are pushed further away. This is done using triplets of entities and schema components, such as fine-tuning SBERT to ensure *total sales* is closer to `gross_sales` and farther from `net_sales`.


DB Schema: create table students{ birthplace, origin, roll_num...

Hints: When user refers to entity hometown use students.birthplace....

Question: Find hometown and roll number of students.

Task: Answer the question by generating a SQL on provided DB schema. **Make sure the output adheres to the hints.**

Figure 3: Prompt template used to provide hints to LLM

7.3 Discussion

User preferences can evolve over time, known as preference drift. For example, a user might initially associate *total sales* with `gross_sales`, but later shift to `final_sales`. Such preference drift can reduce the quality over time. Detecting and adjusting for preference drift is a well-studied in recommender systems literature. A common approach to address this is by maintaining a sliding window of recent user feedback [5], allowing the system to adapt to the most recent interactions. Alternatively, we can use decay functions to gradually reduce the influence of older preferences [12]. Also, human-in-the-loop systems can be employed to manually adjust preferences [34].

User preferences can sometimes be weak, where a user typically favors alternative X (e.g., `gross_sales`) but occasionally prefers Y (e.g., `net_sales`). In such cases, it is best to present both alternatives. A technique that enforces strong preferences would only display X, being the favored option. However, Odin can adapt to both scenarios. In case of a strong preference, Odin can learn to exclude Y, as only X would be marked correct in the calibration set. If it is weakly enforced, both X and Y will be correct in the calibration set. While X may receive a higher score during the selector stage, the cut-off will be set such that both X and Y are included in the final selection, as either could be correct.

8 EVALUATION

We begin by detailing the experimental setup (Section 8.1) and then present the results of an extensive study comparing Odin with various baseline methods. The evaluation reveals several key findings:

- Odin consistently produces a higher-quality SQL result set for user recommendations compared to the baselines. The result sets are not only smaller but also include the correct SQL query more frequently (Section 8.2).
- Even without personalization, Odin outperforms the baselines due to its Generator and Selector components (Section 8.2).
- The schema masking-based Generator in Odin generates a more diverse range of SQL queries compared to both generic and NL2SQL-specific diversity-inducing techniques (Section 8.3).
- Odin's Selector significantly reduces the size of the result set while retaining the correct SQL query. The LLM-based scoring used in this stage proves more effective than SBERT-based scoring (Section 8.4).

8.1 Experimental Setup

Datasets. We use two benchmarks to evaluate our Odin system.

- **AmbiQT Benchmark**: This benchmark is a modified version of the Spider benchmark [36] and addresses different types of ambiguities commonly found in databases (see Fig. 4). In each case, the benchmark modifies the database schema such that there are two correct SQL statements for each question. We use this benchmark to evaluate the Generator component in Odin.
- **Mod-AmbiQT Benchmark**: The original AmbiQT benchmark is not ideal for evaluating personalization, so we created a modified version, Mod-AmbiQT. In this new benchmark, we introduce duplicate columns and tables based on the AmbiQT benchmark. Entities in the questions can map to either of the alternatives, but only one mapping is considered correct. As a result, each question

in this database can now have between 2 to 8 plausibly-correct SQL queries, depending on the number of entities in the question, although only one specific SQL query is designated as correct. This modification is useful for testing personalization.

For example, in the `concert` database, we introduce `artist` and `performers` as two alternatives for the singer table. For the question "*How many singers do we have?*", the two alternative SQL statements could be: "`SELECT COUNT(*) FROM artists`" and "`SELECT COUNT(*) FROM performers`". Out of these, only the first query is considered correct. The entity *singers* consistently maps to `artist` across all questions in the database, allowing us to evaluate personalization.

The new benchmark includes 1,298, 2,148, and 626 Q/A pairs for table, column, and join ambiguity, respectively. We use this benchmark for overall system evaluation.

Kind of ambiguity	Count	Question Text	SQL #1	SQL #2
Column Ambiguity (C)	1240	List the ids of all students.	SELECT roll_number FROM students	SELECT admission_number FROM students
Table Ambiguity (T)	1417	How many singers do we have?	SELECT COUNT(*) FROM artist	SELECT COUNT(*) FROM performer
Join Ambiguity (J)	288	What are the makers and models?	SELECT maker, model FROM model	SELECT t2.maker, t1.model FROM model AS t1 JOIN model_maker AS t2 ON t1.model_id = t2.model_id

Figure 4: Different types of ambiguities in the AmbiQT Benchmark.

Baselines. We compare Odin against two main baselines.

- **Diverse Sampling**: In this baseline, we set the LLM temperature to a high value (*temp*=1.0) to promote the generation of diverse SQL queries. The resulting SQL queries are then presented to the user.
- **Forced Diversity**: In this baseline, we provide the LLM with all previously generated SQL queries and instruct it to produce a new SQL query that differs from the prior ones. The generated SQL queries are then shown to the user.
- **Odin**: The Odin baseline comprises three modules: *Generator*, *Selector*, and *Personalizer* enabled by default, with various components enabled or disabled based on the specific variant. In the Selector, a LLM-based scoring function is employed unless otherwise specified.

All the above baselines are model-agnostic, meaning any LLM can be used. For our evaluation, we utilize Claude 3 Haiku.

To further evaluate the effectiveness of Odin's Generator, we compare it against six baselines from [2] that aim to induce diversity in NL2SQL generation. The goal is to capture all possible ambiguous SQL queries for a given question.

First, we consider a set of naive baselines based on pre-trained language models (PLMs) and LLMs, showcasing their limited ability to generate diverse SQL statements.

1. LLM-X: a commercially available LLM specialized for coding tasks.
2. RESDSQL [16]: one of the top-performing methods on the SPIDER benchmark. We use the 3B variant of RESDSQL, which is the most powerful, for comparison purposes. However, we disable the representation from [6], as it is orthogonal to our approach and could be used alongside it.

Next, we examine baselines that incorporate common sampling techniques to promote more diverse generation. For this, we use the T5-3B checkpoint from the PICARD [23] repository, which fine-tunes T5-3B on the SPIDER dataset. These baselines include:
3. Beam Search (T5-3B-beam): We apply Beam Search with a width of 10 as the default decoding strategy for T5-3B.
4. Top-k Sampling (T5-3B-k): At each decoding step, we sample from the top-50 tokens using top-k sampling with $k = 50$.
5. Nucleus/Top-p Sampling (T5-3B-p): We apply top-p sampling at each decoding step, where tokens that account for 90% of the probability mass are considered, as proposed in [9].
6. Logical Beam: we include this decoding algorithm that navigates the SQL logic space by combining plan based template generation and constrained infilling. For this, we fine-tune a version of Flan T5-3B (with a maximum input length of 512) using the Adafactor optimizer [25] (learning rate 10^{-4}, no decay) on the AmbiQT benchmark, as described in [2].

Metrics: To evaluate the system, we use two key metrics: average accuracy over the workload (*AvgAcc*) and average number of results shown to the user (*AvgResultSize*), both defined in Section 3. *AvgAcc* checks if there exists a SQL query within the result set that matches the execution result of the golden SQL query, and computes the average accuracy over the entire workload. On the other hand, *AvgResultSize* measures the average number of SQL queries presented to the user.

8.2 Overall Evaluation

We first examine how Odin improves accuracy across different types of ambiguities. For this experiment we use the Mod-AmbiQT benchmark. Fig. 5 illustrates the average accuracy of the workload (*AvgAcc*) of the generated results versus the average number of SQL results shown to the user (*AvgResultSize*) for three distinct ambiguity types: join ambiguity, table ambiguity, and column ambiguity. The figure compares Odin with two baselines: Sampling and Forced Diversity.

For all methods, we gradually increase the number of LLM calls K (specifically, we set K to 1, 2, 3, 5, 7, and 10) used for generating SQL queries. As the budget increases, each method shows a higher number of SQL queries to the user, which improves accuracy. Note that for Odin, K refers to the number of LLM calls used in the Generator, i.e., the Generator produces K SQL queries. However, K does not include LLM calls made by the Selector and Personalizer.

Overall Performance: Odin consistently outperforms all other methods, followed by its variants, then Forced Diversity, and finally Sampling. Odin achieves accuracies of 71.6%, 50.3%, and 73.6% for a budget of 10 LLM calls, while displaying an average of 4.9, 4.2, and 3.9 SQL queries to the user for join, table, and column ambiguities, respectively. In contrast, Forced Diversity achieves 33.2%, 38.0%, and 58.8% accuracy across the workload. Compared to Forced Diversity, Odin shows improvements of 38%, 12%, and 15% in accuracy across the three ambiguity types while returning 2-3× fewer SQL queries. The improvement in accuracy is mainly due to the Generator and Personalizer components, while reduction in SQL result set size is attributed to the Selector component. Thus, Odin's result set has up to twice the chance of containing the correct SQL result while being 2-3× smaller in size than Forced Diversity.

Impact of Personalization: We assess the impact of the Personalizer feature, which enables Odin to learn user preferences and

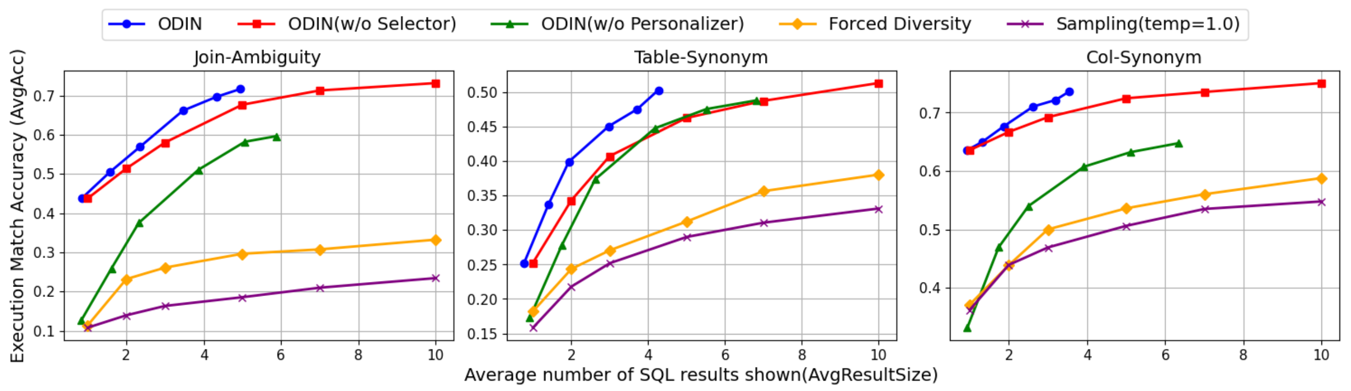


Figure 5: Execution Match Accuracy (*AvgAcc*) versus the average number of results shown to the user (*AvgResultSize*) across three different ambiguity types for various baselines. Odin can achieve up to twice the accuracy while presenting only half the number of SQL queries to the user compared to the next best baseline.

improve performance even at $K = 1$, as shown in Figure 5. At $K = 1$, Odin with Personalizer significantly outperforms Odin without Personalizer by 30%, 8%, and 30% for join, table, and column ambiguities, respectively. This improvement is due to personalized variants better understanding user preferences and generating correct solutions on the first attempt. Personalization can increase the likelihood of finding the correct SQL query in the result set by up to 4-5× for small result set sizes.

Performance Without Personalization: Despite the advantages of Personalizer, Odin still performs effectively without it. For $K = 1$, the accuracy of Odin without Personalization is comparable to that of the baselines. However, as K increases, Odin without Personalization quickly surpasses these baselines. This behavior is expected, as with only one LLM call, all non-personalized baselines generate the same initial SQL query. With a budget of 10 LLM calls, Odin without Personalization outperforms the baselines by 27%, 10%, and 8%. Thus, although Odin without Personalization starts with similar accuracy to the baselines for small result sets, it rapidly improves, achieving up to a 25% gain in accuracy as the set size increases.

Impact of the Selector: We assess the Selector component, which is designed to enhance precision by filtering out incorrect SQL statements while keeping the correct ones, thereby improving precision without significantly affecting recall. Comparing Odin with and without the Selector, we observe that both maintain similar accuracy, with the Selector reducing accuracy by only 1.6%, 1%, and 1.4% for the respective ambiguity types. However, Odin with the Selector displays an average of 5, 4.1, and 3.8 SQL queries per ambiguity type, compared to 10 queries generated by Odin without the Selector. Thus, the Selector component significantly reduces the number of SQL queries displayed by up to 2-2.5× with only a marginal loss in accuracy.

8.3 Effectiveness of Generator

In this experiment, we assess the limitations of traditional diversity-promoting methods in generating ambiguous SQL queries for the NL2SQL task. While these approaches excel at producing a single correct SQL query, subsequent generations often result in minor variations of the same query, missing out on the full spectrum of potential ambiguous interpretations. In contrast, our schema masking-based generation method encourages a broader range of query outputs. To demonstrate this, we conducted evaluations using the AmbiQT benchmark [2], which specifically requires generating two distinct and correct SQL alternatives for each input question. The benchmark measures two metrics: *EitherInTopK*, which checks if at least one of the alternatives is present in the generated set, and *BothInTopK* (Coverage), which evaluates whether both alternatives are present. Two SQL queries are deemed equivalent if their execution results match; thus, verifying the existence of a SQL alternative requires matching its execution results with generated SQL queries.

Across all evaluated baselines, the *EitherInTopK* accuracy remains relatively high, indicating that most methods are capable of producing at least one correct query. However, when the requirement shifts to generating both correct alternatives, the accuracy drops significantly. This trend is clearly illustrated in the results shown in Fig. 6. As hypothesized, traditional diversity-promoting techniques excel at identifying one correct SQL query, evident in high *EitherInTopK* scores, but struggle to generate both correct alternatives, leading to lower *BothInTopK* scores.

Our schema masking approach demonstrates a significant advantage, outperforming the next best baseline, Logical Beam, by factors of 1.9×, 1.2×, and 1.5× in terms of coverage for JOIN ambiguity, Table-Synonyms, and Column-Synonyms, respectively. Additionally, it is noteworthy that different types of ambiguities lead to varying performance across the baselines. LLMs generally handle table-synonyms better than more complex ambiguity types like JOIN ambiguities, as reflected by higher *EitherInTopK* and Coverage scores for table-synonyms. Logical Beam, which employs a specialized decoding strategy to encourage diversity, achieves the second-best performance across most ambiguity types. Nonetheless, the results clearly highlight the superiority of schema masking-based generation, which generates all the ambiguous queries in up to twice as many cases as the next best baseline.

8.4 Selector Ablation Study

The two key components of the Selector are the scoring function and the alpha value. In Fig. 7, we illustrate the trade-off points achieved

Kind of Ambiguity	ODIN	Logical-Beam	LLM-X	RESDSQL	T5-3B-beam	T5-3B-top_k	T5-3B-top_p
EitherInTopK (%)							
Join-Ambiguity	**87.85**	86.81	82.64	68.06	82.64	78.82	76.74
Tbl-Synonyms	**73.96**	71.49	73.68	35.64	64.15	56.60	54.98
Col-Synonyms	**67.44**	68.26	70.57	61.34	60.59	53.50	53.50
BothInTopK (Coverage) (%)							
Join-Ambiguity	**58.33**	31.60	31.94	9.03	13.89	1.39	0.69
Tbl-Synonyms	**59.99**	51.52	38.39	10.66	40.08	13.34	10.94
Col-Synonyms	**43.69**	29.10	9.73	13.77	12.12	2.64	2.56

Figure 6: ODIN's Generator results alongside various SQL generation baselines for ambiguity types such as Join, Table Synonyms, and Column Synonyms. Each question has two SQL alternatives, and we assess whether either or both are in the generated result set (EitherInTopK and BothInTopK). While most baselines achieve high EitherInTopK accuracies by generating at least one correct alternative, ODIN outperforms them by generating both alternatives in up to twice as many cases as the next best baseline.

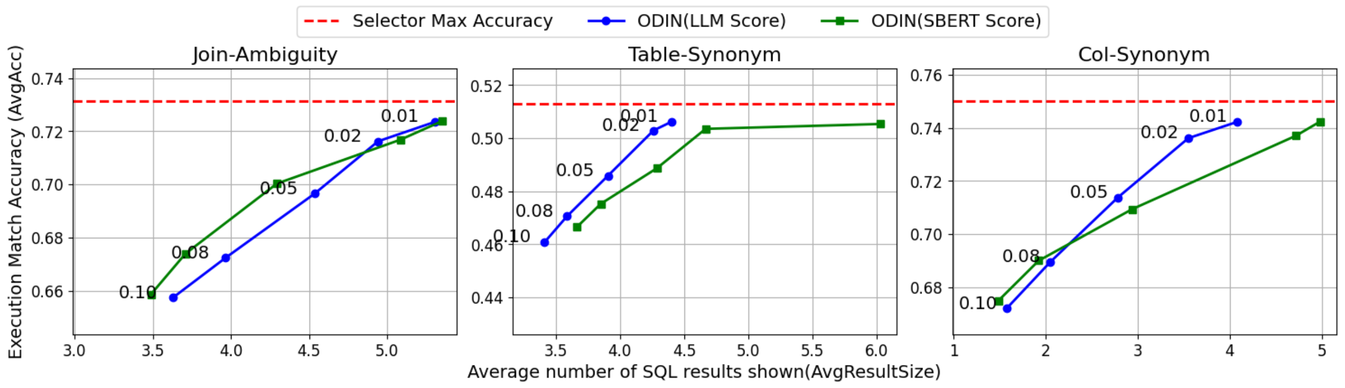


Figure 7: ODIN's Execution Match Accuracy (*AvgAcc*) versus the average number of results shown to the user (*AvgResultSize*) on across three ambiguity types, using various Selector scoring functions (LLM-based and SBERT-based). For each baseline, we vary the Selector's alpha value from 0.01 to 0.1. Higher alpha values lead to more SQL queries being discarded by the Selector, which decreases both accuracy and result set size.

by modifying the scoring function and adjusting the alpha values from 0.01 to 0.1 for each scoring function. For this experiment, we utilize two scoring functions: one based on an LLM and the other based on SBERT. Additionally, we include the maximum possible accuracy of the Selector, represented by the accuracy of ODINwithout the Selector.

Impact of Scoring Function: An effective scoring function assigns low scores to correct SQL queries and high scores to incorrect ones, improving result pruning. For an alpha of 0.01, the LLM-based scoring displays an average of 5.3, 4.4, and 4.1 SQL queries across three ambiguity classes, while SBERT averages 5.35, 6.1, and 5.9. The LLM-based scoring presents 1.5 to 2 fewer SQL queries than SBERT, although the difference is less pronounced in the case of join ambiguity. While we expect the LLM to outperform SBERT due to its larger model size, SBERT performs better in scenarios where names of ambiguous tables and columns are nearly identical. For example, in a join ambiguity involving the `birthplace` column in the `students` table and the similarly named `birthplace` column in the `student_birthplace`, SBERT assigns a similarity score close to 1.0, leading to higher scores for such ambiguous queries. However, SBERT struggles with cases involving synonyms, where the names are less similar.

Impact of Alpha: The alpha value represents the maximum allowable reduction in accuracy when pruning SQL results from the generation stage. Larger alpha values enable the Selector to prune more SQL queries, resulting in a smaller result set but lower accuracy. Across all three ambiguity classes, increasing the alpha value correlates with a decrease in both accuracy and the number of SQL queries displayed. Higher alpha values allow for a greater drop in recall, leading to stricter thresholds and fewer SQL queries shown to the user. For an alpha value of 0.1, the LLM-based scoring achieves accuracies of 65.7%, 46.1%, and 67.2%, while maximum accuracies are 73.2%, 51.3%, and 75.0%, respectively. Notably, the accuracy with an alpha of 0.1 is approximately 10% lower than that without the Selector, reflecting the trade-off that the Selector is designed to manage.

9 CONCLUSION

In this paper, we introduced ODIN, an NL2SQL system that handles schema ambiguity. ODIN further personalizes SQL results based on user feedback. ODIN demonstrates consistent accuracy improvements across different ambiguity types in our benchmarks.

REFERENCES

[1] Anastasios N Angelopoulos, Stephen Bates, et al. 2023. Conformal prediction: A gentle introduction. *Foundations and Trends® in Machine Learning* 16, 4 (2023), 494–591.
[2] Adithya Bhaskar, Tushar Tomar, Ashutosh Sathe, and Sunita Sarawagi. 2023. Benchmarking and improving text-to-sql generation under ambiguity. *arXiv*

preprint arXiv:2310.13659 (2023).

[3] Han Fu, Chang Liu, Bin Wu, Feifei Li, Jian Tan, and Jianling Sun. 2023. Catsql: Towards real world natural language to sql applications. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1534–1547.

[4] Matthieu Futeral, Cordelia Schmid, Ivan Laptev, Benoît Sagot, and Rachel Bawden. 2023. Tackling Ambiguity with Images: Improved Multimodal Machine Translation and Contrastive Evaluation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 5394–5413.

[5] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A survey on concept drift adaptation. *ACM computing surveys (CSUR)* 46, 4 (2014), 1–37.

[6] 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. *arXiv preprint arXiv:2109.05153* (2021).

[7] Zihui Gu, Ju Fan, Nan Tang, Lei Cao, Bowen Jia, Sam Madden, and Xiaoyong Du. 2023. Few-shot Text-to-SQL Translation using Structure and Content Prompt Learning. *Proc. ACM Manag. Data* 1, 2, Article 147 (jun 2023).

[8] Zihui Gu, Ju Fan, Nan Tang, Lei Cao, Bowen Jia, Sam Madden, and Xiaoyong Du. 2023. Few-shot text-to-sql translation using structure and content prompt learning. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–28.

[9] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751* (2019).

[10] Bairu Hou, Yujian Liu, Kaizhi Qian, Jacob Andreas, Shiyu Chang, and Yang Zhang. 2024. Decomposing Uncertainty for Large Language Models through Input Clarification Ensembling. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*.

[11] George Katsogiannis-Meimarakis and Georgia Koutrika. 2023. A survey on deep learning approaches for text-to-SQL. *VLDB J.* 32, 4 (2023), 905–936.

[12] Ralf Klinkenberg. 2004. Learning drifting concepts: Example selection vs. example weighting. *Intelligent data analysis* 8, 3 (2004), 281–300.

[13] Lorenz Kuhn, Yarin Gal, and Sebastian Farquhar. 2022. CLAM: Selective Clarification for Ambiguous Questions with Large Language Models. *CoRR* abs/2212.07769 (2022).

[14] Dongryeol Lee, Segwang Kim, Minwoo Lee, Hwanhee Lee, Joonsuk Park, Sang-Woo Lee, and Kyomin Jung. 2023. Asking Clarification Questions to Handle Ambiguity in Open-Domain QA. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. 11526–11544.

[15] Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024. The Dawn of Natural Language to SQL: Are We Fully Ready? *Proc. VLDB Endow.* 17, 11 (aug 2024), 3318–3331.

[16] Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023. Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 13067–13075.

[17] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024. CodeS: Towards Building Open-source Language Models for Text-to-SQL. *Proc. ACM Manag. Data* 2, 3, Article 127 (may 2024).

[18] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024. Codes: Towards building open-source language models for text-to-sql. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–28.

[19] 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. arXiv:2305.03111 [cs.CL]

[20] Yihang Li, Shuichiro Shimizu, Weiqi Gu, Chenhui Chu, and Sadao Kurohashi. 2022. VISA: An Ambiguous Subtitles Dataset for Visual Scene-aware Machine Translation. In *Proceedings of the Thirteenth Language Resources and Evaluation Conference*. 6735–6743.

[21] Abdul Quamar, Vasilis Efthymiou, Chuan Lei, and Fatma Özcan. 2022. Natural Language Interfaces to Data. *Found. Trends Databases* 11, 4 (2022), 319–414.

[22] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, Hong Kong, China, 3982–3992. https://doi.org/10.18653/v1/D19-1410

[23] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. arXiv:2109.05093 [cs.CL]

[24] Glenn Shafer and Vladimir Vovk. 2008. A tutorial on conformal prediction. *Journal of Machine Learning Research* 9, 3 (2008).

[25] Noam Shazeer and Mitchell Stern. 2018. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*. PMLR, 4596–4604.

[26] Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. CHESS: Contextual Harnessing for Efficient SQL Synthesis. *CoRR* abs/2405.16755 (2024).

[27] Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. CHESS: Contextual Harnessing for Efficient SQL Synthesis. arXiv:2405.16755 [cs.LG] https://arxiv.org/abs/2405.16755

[28] Janette Vazquez and Julio C Facelli. 2022. Conformal prediction in clinical medical sciences. *Journal of Healthcare Informatics Research* 6, 3 (2022), 241–252.

[29] Vladimir Vovk, Alexander Gammerman, and Glenn Shafer. 2005. On-line compression modeling I: Conformal prediction. *Algorithmic learning in a random world* (2005), 189–221.

[30] Bing Wang, Yan Gao, Zhoujun Li, and Jian-Guang Lou. 2023. Know What I don't Know: Handling Ambiguous and Unknown Questions for Text-to-SQL. In *Findings of the Association for Computational Linguistics: ACL 2023*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). 5701–5714.

[31] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Qian-Wen Zhang, Zhao Yan, and Zhoujun Li. 2023. MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL. *CoRR* abs/2312.11242 (2023).

[32] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Qian-Wen Zhang, Zhao Yan, and Zhoujun Li. 2023. Mac-sql: Multi-agent collaboration for text-to-sql. *arXiv preprint arXiv:2312.11242* (2023).

[33] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2021. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. arXiv:1911.04942 [cs.CL]

[34] Xingjiao Wu, Luwei Xiao, Yixuan Sun, Junhang Zhang, Tianlong Ma, and Liang He. 2022. A survey of human-in-the-loop for machine learning. *Future Generation Computer Systems* 135 (2022), 364–381.

[35] Miao Xiong, Zhiyuan Hu, Xinyang Lu, Yifei Li, Jie Fu, Junxian He, and Bryan Hooi. 2023. Can llms express their uncertainty? an empirical evaluation of confidence elicitation in llms. *arXiv preprint arXiv:2306.13063* (2023).

[36] 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. arXiv:1809.08887 [cs.CL]