SQUAB: Evaluating LLM robustness to Ambiguous and Unanswerable Questions in Semantic Parsing

Simone Papicchio [α,β] **Luca Cagliero**[α] **Paolo Papotti**[β]
[α]Politecnico di Torino, Italy [β]EURECOM, France
{simone.papicchio, luca.cagliero}@polito.it papotti@eurecom.fr

Abstract

Large Language Models (LLMs) have demonstrated robust performance in Semantic Parsing (SP) for well-defined queries with unambiguous intent and answerable responses. However, practical user questions frequently deviate from these ideal conditions, challenging the applicability of existing benchmarks. To address this issue, we introduce SQUAB, an automatic dataset generator of Ambiguous and Unanswerable questions. SQUAB generates complex, annotated SP tests using a blend of SQL and LLM capabilities. Results show that SQUAB reduces test generation costs by up to 99% compared to human-based solutions while aligning with real-world question patterns. Furthermore, these tests challenge LLM performance while revealing disparities between public and proprietary datasets. This highlights the need for a dynamic, automatic dataset generator as SQUAB. The code is designed for user extension to accommodate new ambiguous and unanswerable patterns and is available at https://github.com/spapicchio/squab.

1 Introduction

Models reveal vulnerabilities when user interactions introduce linguistic ambiguity and unanswerable questions. Ambiguity, in forms like vague references or unclear scope, leads to multiple valid query interpretations in Semantic Parsing (SP), also known as Text2SQL (Saparina and Lapata, 2024; Bhaskar et al., 2023). Unanswerable queries arise when required information is missing or out-of-scope (Wang et al., 2023). These complexities hinder model effectiveness in real-world deployments, where proprietary data and unpredictable inputs diverge from controlled benchmarks.

The gap between benchmark performance and real-world applicability calls for realistic model evaluation. Our solution automatically generates comprehensive tests, including NL questions and their corresponding SQL queries. Unlike recent approaches that rely on manually crafted tests (Saparina and Lapata, 2024), our method requires only a dataset from the end-user. The framework autonomously creates diverse tests - such as ambiguous columns and out-of-scope functions - covering real-world complexities. As data, language, and models evolve, end-users can employ these tests to determine the best solution with minimal cost.

Our framework, SQUAB[1], supports six ambiguity and unanswerability categories for SP, as shown in Table 1. For example, for *Column ambiguity*, our solution generates a question asking for the "field goal" of a player, potentially referring to FG% or 3FG%. For unanswerability, it might ask for a player's "number of fouls" when this attribute is absent from the schema.

SQUAB enables users to input enterprise data to autonomously generate both ambiguous and unanswerable annotated tests. By combining query generation with LLM-driven steps, SQUAB produces realistic, challenging tests at minimal cost. These tests are then processed by a target LLM, allowing SQUAB to assess its robustness to ambiguity and unanswerable questions. This fully automated process streamlines complex test creation while providing detailed LLM performance assessments.

The six categories covered by SQUAB extend beyond those explored in prior SP benchmarks, as shown in Table 2. Previous works face limitations: ambiguity tests often rely on predefined templates (Wang et al., 2023; Papicchio et al., 2024) or hand-crafted examples (Saparina and Lapata, 2024; Bhaskar et al., 2023). They typically focus on column vagueness or a limited ambiguity types, overlooking unanswerable queries. Moreover, most benchmarks do not differentiate between proprietary and general-purpose datasets, ignoring their performance differences. SQUAB bridges the gaps

[1]**SQL** **U**nanswerable and **A**mbiguous **B**enchmarking

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 17938–17958

Category	Definition	Examples
Column Ambiguity	Question mentions an entity that refers to multiple table attributes.	**Question**: List the *field goals* percentages of each player. **Explanation**: *field goals* may refer to either FG% or 3FG%
Scope	The question has a dual interpretation of quantifiers, specifically in their collective versus distributive readings.	**Question**: List the player that *every* team supports. **Explanation**: The question can mean either 'the player supported by all teams' or 'for each team, the supported player'
Attachment	In the question, it is unclear how a modifier or phrase is attached to the rest of the sentence	**Question**: List players and teams with 3FG% *above 50%* **Explanation**: It is unclear whether the constraint *above 50%* must be enforced on all the players of a team or each player separately
Column Unanswerable	The question mentions an entity that cannot be referred to any of the attributes in the table.	**Question**: What is the number of *Fouls* for LeBron James? **Explanation**: The attribute *Fouls* is not present in the table schema.
Calculation Unanswerable	The question contains an undefined UDF executable in SQL but not defined in the DBMS.	**Question**: What is the *efficiency score* of LeBron James? **Explanation**: The *efficiency score* can be calculated in SQL, but it is not defined.
Out Of Scope	The question requires the execution of a function outside the scope of SQL.	**Question**: What will be the final score of the Chicago Bulls' upcoming game? **Explanation**: Forecasting functionalities are not supported by the standard SQL language.

Table 1: Example of ambiguous and unanswerable questions. The examples are based on the table Players(*Player*, *Team*, FG%, 3FG%) where FG% and 3FG% respectively indicate the 2-Point and 3-point Field Goal percentages.

	Column Amb.	Attach	Scope	Column Unans.	Calculation Unans.	Out Of Scope
Damber	✓	✗	✗	✗	✗	✗
AmbiQT	✓	✗	✗	✗	✗	✗
Ambrosia	✓	✓	✓	✗	✗	✗
NoisySP	✓	✗	✗	✓	✳	✳
ArcherFish	✳	✗	✗	✳	✳	✳
SQUAB	✓	✓	✓	✓	✓	✓

Table 2: Works about Ambiguity and Unanswerable questions: ✓ denotes defined and included in the dataset; ✳ denotes defined but not included; ✗ denotes missing.

in previous work by exploring three research questions: How effectively do generated tests mimic real-world ambiguous query patterns compared to human-crafted datasets? Can SQUAB balance cost and quality relative to manual test generation? To what extent do these query patterns challenge LLM performance, and how do results differ between public benchmarks and proprietary datasets, which often reveal significant performance gaps?

Our experiments provide comprehensive answers to these questions. SQUAB effectively aligns with human-crafted datasets. By generating synthetic tests with up to 99% cost savings over manually crafted tests, SQUAB achieves close evaluation results across four LLMs, particularly in *Scope* and *Attachment* ambiguities. This confirms that SQUAB captures real-world question complexities, acting as an effective proxy for expensive human-curated evaluations. SQUAB's tests reveal notable LLM performance discrepancies between open benchmarks and proprietary data, emphasizing the need for tailored evaluations. It provides robust, adaptable assessments, advancing LLM robustness and adaptability to real-world applications.

2 Preliminaries

Semantic Parsing. Let $\mathcal{D}$ be a database, with schema $\mathcal{D}_\mathcal{S}$, consisting of the relational tables T_1, T_2, ..., $T_{|\mathcal{D}|}$, Q=$\{q_1,q_2,...,q_N\}$ be the set of *non-equivalent*[2] SQL queries that can be formulated on $\mathcal{D}$, $\mathcal{P}(Q)$ be the power set of Q, and nl a natural language question. The goal of the Semantic Parsing task is to leverage a Language Model to learn a deterministic mapping f: $nl \rightarrow \mathcal{P}(Q)$ To this end, the LM is fed with the tables' schema and, optionally, the instances in $\mathcal{D}$ (Floratou et al., 2024). With not ambiguous and answerable questions, the cardinality of f is 1.

Ambiguous Questions. A question is *ambiguous* if it can be mapped to two or more *non-equivalent* SQL queries i.e. $|f(nl)| > 1$. We refer to these queries as SQL interpretations.

We focus on existing definitions of ambiguity as summarized in Table 2, and propose a method for generating ambiguity for each T in $\mathcal{D}$.

Column ambiguity occurs when the nl has multiple correct interpretations based on different attributes in T, e.g., *field goals* can refer to FG% or 3FG%.

Scope ambiguity occurs when it is unclear which

[2]Non-equivalent queries produce different results.

elements a quantifier, such as *each*, *every*, or *all*, refers to (Kiss, 2006). In the example in Table 1, the quantifier *every* may be interpreted widely (all the *players*), or narrowly (each *player* separately).

Attachment ambiguity is when it is unclear how a phrase is attached to the rest of the sentence (Resnik, 1993), leading to multiple interpretations. In Table 1, the phrase *above 50%* may apply to all team players or each player individually.

Unanswerable questions. A question is *unanswerable* if $f(nl) = \emptyset$, i.e., there is no valid mapping between the nl and $P(Q)$. SQUAB automatically generates unanswerable questions that align with existing definitions (Wang et al., 2023).

Column unanswerable is when an attribute in nl has no relationship with any attribute in T. In Table 1, attribute *Foul* does not exist in the table.

Calculation unanswerable is when nl references an aggregated value that depends on a user-defined function (UDF) not defined in $\mathcal{D}$, e.g., the *efficiency score* in Table 1.

Out-of-scope is when nl references a UDF that is out of the scope of standard SQL, e.g., the forecasting request in Table 1.

3 Methodology

SQUAB consists of three key steps, as shown in Figure 1. While a single call to an LLM could generate the questions, our experimental results indicate that SQUAB's design maximizes the variability of the questions while reducing inference costs and aligning more closely with human-crafted datasets. For the six categories, we instantiated the three steps in our pipeline by defining the logic programmatically. This operation has to be done once for each category and enables test generation at scale for every new, unseen input table.

To illustrate our approach, we use an example in Table 3. Starting from table *Limits*[3] , we want to obtain *Column Ambiguity* questions and the corresponding SQL interpretations.

(i) Pattern Identification (PI) Let $T \in \mathcal{D}$ be a table with attributes $A = \{a_n \mid n \in [1, N]\}$, where each attribute includes its corresponding values. This step identifies patterns $P \subseteq A$ that could lead to ambiguous or unanswerable questions, where each pattern $p \in P$ consists of a subset of table attributes, denoted as $\{a_k \mid k \in [1, K],\ a_k \in A\}$ - the same attribute may appear in multiple patterns. In our example, generation begins by identifying semantically similar attributes, as these are the ones most likely to lead to an ambiguous question. In the example $p = \{BagWeight, BagPieces\}$. As another example, for the *Scope* category, the PI step identifies columns related by a many to many relationships in the data.

Column Ambiguity

Table: Limits(Airlines, *BagWeight*, *BagPieces*, Date)

PI: Semantically Close Attributes: [*BagWeight*, *BagPieces*]

RM: hypernym: baggage limit

SQL interpretations:

```
- SELECT Airlines, MAX("BagWeight") FROM Limits
    GROUP BY Airlines;
- SELECT Airlines, MAX("BagPieces") FROM Limits
    GROUP BY Airlines;
```

Question: What is the maximum *baggage limit* for each airline?

Table 3: SQUAB generation for *Column ambiguity*.

(ii) Relational Metadata (RM) one approach would be to pass p to an LLM with a prompt to generate the $\langle nl_i, answer_i \rangle$ pair, hereafter defined test. However, this method produces results with less variability and simpler questions.

To better steer the generation, before generating the NL question, we enrich the pattern with information for the category at hand. In the example, to construct a question like "What are the total *baggage limits* for each airline?", we enrich p with the ambiguous entity *baggage limits*, which may refer to *BagWeight* or *BagPieces*.

We therefore use this second step to obtain *Relational Metadata*, which (i) is derived from an underlying relationship among the attributes in every p and (ii) it enriches the pattern. In our example, this relationship the generalization of the attribute labels with a *hypernym*, i.e., a word that encompasses the meaning of a group of related words. As another example, for a *Column Unanswerable* test, this step generates the label of a relevant attribute that is not in the input table, e.g., *LiquidsLimit*.

Given a pattern p, this step produces an enriched representation $p^* = \varphi(p)$, where φ is a function that generates the *RM*. In SQUAB, φ can be defined through an LLM call or with heuristics. If the underlying relationship does not exist, the generation process from p terminates at this step. This process is repeated for every $p \in P$.

[3]We focus on single-table scenarios in our framework. This approach allows a clear evaluation of LLMs' ability to handle the linguistic ambiguities without the confounding factor of complex multi-table joins. We show experimentally that the challenges posed by ambiguity and unanswerability types is effectively surfaced in a single-table context.

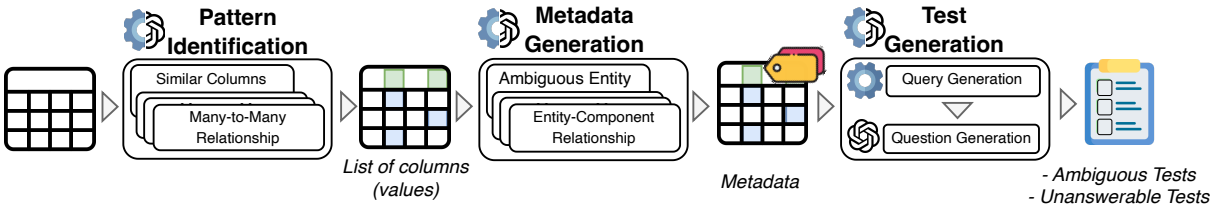

Figure 1: Given a table, SQUAB executes a pipeline based on *Pattern Identification*, *Metadata Generation*, and *Test Generation* to produce SP tests. denotes the use of LLM and the use of scripts.

(iii) Test Generation (TG) At this step, we have the components to generate the NL question (nl) and its answer, i.e., the set of SQL interpretations of the ambiguous nl.

Again, we do not pass p^* directly to an LLM to generate a test because we require finer control over query generation to introduce ambiguity or unanswerability in specific parts of the nl. For instance, in our example, the ambiguity arises in an SQL aggregation function such as 'SUM', allowing for greater variability and more granular test cases.

To enforce a precise control over the SQL constructs and the complexity of the test, we pivot our generation process on the queries for two reasons. First, they can be easily generated starting from the table with templates at no cost (Papicchio et al., 2023). Second, they precisely guide the text generation process as they provide declarative specifications. Therefore, SQUAB first generates (unambiguous, answerable) SQL queries based on the attributes in the enriched pattern, then builds on top of those to generate the final nl with an LLM.

For each extended pattern p^*, this step generates a set of ambiguous or unanswerable SP tests. A test is a pair $\langle nl_i, label_i \rangle$, where nl_i is a NL question, and $label_i$ consists of the SQL interpretations of the ambiguous nl_i or of a placeholder indicating its unanswerability. The process of generating nl_i consists of three steps.

1. SQL Query Generation. Let $Q_a \subseteq Q$ be the set of non-equivalent queries over $\mathcal{D}$ that contain one randomly selected attribute $a \in p$. This step employs a function $\tau : (T, a) \rightarrow Q_a$ which generates all SQL queries involving a in T. These queries are generated with templates and serve as the foundation for generating the ambiguous/unanswerable queries next. Notably, a may appear alongside other attributes and in various SQL clauses, including 'GROUP BY', 'HAVING', 'MIN', etc. In our example, this step produces the query u:

```
SELECT Airlines, MAX("BagWeight")
FROM Limits GROUP BY Airlines
```

where attribute a is *BagWeight*.

2. Query Transformation. Each query in Q_a acts as a seed for an ambiguous or unanswerable query. Given a query $\hat{q} \in Q_a$, this step applies a transformation function ω using p^* with its RM: $\omega : (\hat{q}, p^*) \rightarrow \hat{Q}$ where $\hat{Q}$ represents the transformed version(s) of $\hat{q}$. For ambiguous tests, $\hat{Q}$ comprises possible SQL interpretations. For example, given the query u above, we use the attribute in the pattern to obtain a second query u':

```
SELECT Airlines, MAX("BagPieces")
FROM Limits GROUP BY Airlines
```

In case of unanswerable tests, $\hat{Q}$ corresponds to one unsolvable SQL query, for example:

```
SELECT Airlines, MAX("LiquidsLimit")
FROM Limits GROUP BY Airlines
```

3. Question Generation. The question nl is generated based on the queries in $\hat{Q}$, the extended pattern p^*, and the schema $\mathcal{D}_\mathcal{S}$ of $\mathcal{D}$:

$$nl = \mathcal{L}(prompt, P^*, \hat{Q}, \mathcal{D}_\mathcal{S})$$

where $\mathcal{L}$ is a large language model (LLM) and *prompt* refers to predefined prompts for generating ambiguous and unanswerable questions. The prompts used in SQUAB are provided in Appendix E.

3.1 Generation of Ambiguous Tests

We outline the generation of ambiguous tests with SQUAB. The final TG step follows the general procedure for all categories. For *Column ambiguity*, we automatically generate template queries Q_a (Papicchio et al., 2023), and for *Scope* and *Attachment* ambiguity, we use established templates (Saparina and Lapata, 2024). The tests with empty SQL interpretations are discarded.

Column Ambiguity. As we discussed this category in our example, we only add details here. In *PI*, each pattern p is a cluster of attributes based on the cosine similarity of the labels' embeddings[4]. For each pattern p, we prompt the LLM to obtain a hypernym (Manning and Schütze, 2001).

[4]We use text-embedding-3-large with threshold $\theta = 0.7$

Scope

Table: Players(*Player*, *Team*, FG%, 3FG%)
***PI*:** Many-to-Many relationship: *Player*, *Team*
***RM*:** Entity: *Team*; Component: *Player*
SQL Interpretations:

```
- SELECT DISTINCT "Player", "Team" FROM Players;
- SELECT "Player" FROM Players GROUP BY "Player"
    HAVING COUNT(DISTINCT "Team")= (SELECT
    COUNT(DISTINCT "Team") FROM Players);
```

Question: List the *Player* every *Team* support.

Table 4: Generation process for *Scope* ambiguity.

Attachment

Table: Accommodations(Name, Location Type:: *Hotels, Pods*, Price:: *200*)
PI: Distinct value in *Type* have same value in *Price*
RM *Hotels* and *Pods* implicitly represent *Name*.
SQL interpretations

```
- SELECT Name FROM Accommodations WHERE Type =
    "Hotels" OR Type = "Pods" AND Price = "200";
- SELECT Name FROM Accommodations WHERE (Type =
    "Hotels" OR Type = "Pods") AND Price = "200";
```

Question: List *hotels* and *pods* priced at *200*.

Table 5: Generation process for *Attachment* ambiguity.

Column Unanswerable

Table: Customers(id, Name, Age, Region)
PI: Table Schema: id, Name, Region
RM: New Attribute: customer_segment, Type: categorical
SQL query:

```
SELECT Region, COUNT("customer_segment")
FROM Customers GROUP BY Region;
```

Question: How many customer segments exist per region?

Table 6: Generation process for *Column Unanswerable*.

Calculation Unanswerable

Table: Credits(id, Age, Income, Credit_score)
PI: Table Schema: id, Age, Income, Credit_score.
RM: UDF: interest_rate('Age', 'Income', 'Credit_score'),
UDF output type: "numerical",
UDF python code: $(balance * 0.05) + (credit\_score * 0.02) - (loan\_history * 0.01)$
SQL Query:

```
SELECT AVG("interest_rate('Age', 'Income',
    'Credit_score')") FROM Credits;
```

Question: Calculate the average interest rate.

Table 7: Generation process for *Calcul. Unanswerable*.

Scope. *Scope* ambiguity arises from the dual interpretation of quantifiers, Table 4 provides an example of its generation. The *PI* step programmatically identifies attribute pairs with a many-to-many relationship, as quantifiers cannot be used otherwise. In the *RM* step, an LLM checks the presence of an entity-component relationship, e.g., *Team* is classified as entity and *Player* as component. This identifies a *Scope* ambiguity since the interpretations are: (i) whether the question refers to components collectively with their entity (*the list of players along with their team*), (ii) or individually across all entities (*the player supported by all teams*).

Attachment. *Attachment* ambiguity arises when syntactic structures make it unclear what the subject of a modifier is. In the generated question in Table 5, it is not clear if the modifier 200 refers to *pods* or both *hotels* and *pods*. The subjects and the modifier are attribute values derived from two attributes. The *PI* step identifies attribute pairs where different values (representing potential subjects) share the same value (the quantifier) for another attribute, as *AccommodationType* and *Price* in our example. To build the nl, the quantifier's subject values must implicitly match the projected attribute. For example, *Hotels* and *Pods* implicitly refer to the *Name* attribute in our example. We detect this relationship using a simple heuristic: the projection attribute must include a canonical entity label like 'Name', and the quantifier attribute must be categorical.

3.2 Generation of Unanswerable Tests

To our knowledge, SQUAB is the first framework to create unanswerable nl automatically. We use a simplified approach that excludes multi-attribute cases, using the table schema as the PI pattern. In the TG step, template queries Q_a are generated from templates and transformed via the RM into $\hat{Q}$. Queries in $\hat{Q}$ that are executable in D are discarded to ensure generated questions are unanswerable.

Column Unanswerable. This category occurs when a question refers to an attribute that does not have an association with any attribute in D, as in the example in Table 6. In the RM step, the LLM uses the table schema to generate a non-existing attribute label that aligns with the table context and its type. For instance, the LLM might generate the categorical attribute *customer_segment*. Subsequently, the column type is employed in the TG step to appropriately integrate the new attribute label into the SQL query, e.g., ensuring that a categorical column is not used in an AVG operation.

Calculation Unanswerable. This category arises when a question includes a SQL-executable UDF whose internal procedure is unspecified (Table 7). In the RM step, an LLM generates not only the UDF call in the SQL query, such as 'interest_rate('Age', 'Income', 'Credit_score')', but also the UDF's output type and its corresponding code. Like in *Column Unanswerable*, the output type is

Out Of Scope

Table: Credits(id, Age, Income, Credit_score)
PI: Table Schema: id, Age, Income, Credit_score
RM: UDF: predict_interest_rate('Age', 'Credit_score'), UDF output type: "numerical"
SQL Query:

```
SELECT Income FROM Credits GROUP BY Age HAVING
    predict_interest_rate('Age','Credit_score')>5;
```

Question: What is the income for age groups with a predicted interest rate over 5?

Table 8: Generation process for *Out-Of-Scope* as forecasting is beyond the SQL standard.

used to transform the SQL templates correctly. The UDF code enables the verification of the SQL executability: without the UDF defined for D, the transformed query $\hat{Q}$ must fail.

Out-Of-Scope. In this case the UDF is outside the scope of SQL, such as "making a prediction" in the example in Table 8. In the RM step, the LLM is prompted to generate the UDF call and its output type for the SQL query, but not the code.

4 Experiments

Datasets. We evaluate SQUAB on Ambrosia (Saparina and Lapata, 2024) and Beaver (Chen et al., 2024). Ambrosia is a benchmark with databases that are designed to incorporate ambiguities, whereas Beaver is an enterprise database with 99 tables. To balance costs with statistical significance, we sample 33% of Beaver's tables. For unanswerable tests, we select the top 33 Ambrosia tables by decreasing arity. More details in Appendix A.

Test Generation Baselines. We compare SQUAB to two baseline methods for ambiguous test generation: *Ambrosia* and *All-LLM*. For unanswerable tests, we compare only with *All-LLM*, since, to our knowledge, no dataset covers all three unanswerable categories. We generate all tests using *gpt-4o-2024-11-20* for SQUAB and All-LLM. Ambrosia includes human-crafted tests for all ambiguity types, which we use to compare against SQUAB generated tests. We restrict Ambrosia tests to single-table queries for *Column* and *Attachment* ambiguities. For *Scope ambiguity*, we compare Ambrosia's multi-table tests with our single-table versions generated from the denormalized Scope database. All-LLM prompts the LLM with table context, a category description, and three examples, without SQUAB's structured generation pipeline. More details on the prompt used and variability of the generated questions are in Appendix E and Appendix D, respectively.

SP LLMs. We test the ambiguous and unanswerable tests with the following LLMs: proprietary GPT-4o-mini (gpt-4o-mini-2024-07-18) and Gemini-1.5-pro models, and open-source 8B and 70B versions of Llama 3.1 Instruct model. For ambiguous queries, we follow the prompting format of Ambrosia (Saparina and Lapata, 2024). For unanswerable queries, models are instructed to return 'NOT ANSWERABLE'. All evaluations provide models with database schema and content. Beaver rows per table (cardinality) are limited to 10 rows for efficiency; Ambrosia tables are used fully.

Evaluation Metrics. For ambiguous tests, we report the F1-score calculated between the SQL interpretations and the predicted queries matched by execution accuracy (Li et al., 2024a), following previous work (Saparina and Lapata, 2024). For unanswerable tests, we measure accuracy as the percentage of correctly identified "NOT ANSWERABLE" responses.

Generation Cost Calculation. The generation cost is measured in dollars. For Ambrosia, we derive the cost from its appendix. For SQUAB and All-LLM, we calculate the cost based on the number of input and output tokens used in each LLM call, multiplied by the respective model's pricing. The total cost is the sum of costs incurred across all generated tests.

4.1 Results and Discussion

Hereafter we present the main results by separately addressing our main research questions.

Can synthetic tests mimic human-crafted ambiguous query patterns? Table 9 compares human-generated (Ambrosia), SQUAB-generated, and All-LLM-generated ambiguous tests on cost, test counts, LLM F1-scores, and Kendall's τ for LLM ranking correlation.

The automatically generated tests achieve F1-Scores comparable to those of human-curated ones, but at a significantly lower inference cost (up to 99% cost reduction, e.g., SQUAB $4 vs. Ambrosia $1105). The SQUAB rankings are mostly in agreement with those of Ambrosia (τ>0.6 for all ambiguity categories). Conversely, All-LLM and Ambrosia exhibit a weaker correlation because the LLM-based approach fails to capture the nuanced query structures and variations inherent in ambiguous data. Overall, the results confirm the

Test Type	$ Cost (↓) Human	SQUAB	All-LLM	# tests Human	SQUAB	All-LLM	Models	% F1-score (↑) Human	SQUAB	All-LLM	Kendall τ SQUAB	All-LLM
Scope	$769	$2.30	$3.05	485	442	411	Gpt4o-mini	0.17	0.38	0.35	**1.0**	0.66
							llama3.1-8b	0.03	0.17	0.17		
							llama3.1-70b	**0.38**	**0.43**	0.57		
							Gemini 1.5 pro	0.23	0.39	**1.00**		
Attach	$153	$0.67	$0.90	97	153	152	Gpt4o-mini	0.15	0.25	0.21	**0.92**	0.33
							llama3.1-8b	0.13	0.23	0.17		
							llama3.1-70b	0.12	0.23	0.31		
							Gemini 1.5 pro	**0.55**	**0.59**	**0.48**		
Column Ambiguity	$183	$0.30	$0.42	109	110	112	Gpt4o-mini	0.35	0.22	0.27	**0.67**	**0.67**
							llama3.1-8b	0.30	0.26	0.34		
							llama3.1-70b	**0.46**	**0.36**	**0.40**		
							Gemini 1.5 pro	0.41	0.30	0.36		
Total	$1105	$4.53	$4.35	698	713	675	**Average**	0.27	0.32	0.39	0.75	0.55

Table 9: Comparing human-generated (Ambrosia), SQUAB-generated, and All-LLM-generated tests assesses how synthetic tests mirror human-based ones. Annotation costs for Ambrosia derived from its appendix. The Kendall τ is calculated between the ranking of LLMs by F1-score on Ambrosia tests and that on synthetic tests (by SQUAB and All-LLM, respectively) for each ambiguity type.

SQUAB's effectiveness in serving as a proxy for more expensive, human-curated evaluations.

Test Type	$ Cost SQUAB	all LLM	# tests SQUAB	all LLM	Models	%F1-score SQUAB	all LLM
Scope	$0.53	$9.63	32	14	Gpt4o-mini	**0.16**	0.34
					llama 3.1 8b	0.01	0.30
					llama 3.1 70b	0.04	0.26
					Gemini 1.5	0.07	**0.31**
Attach	$0.55	$6.60	34	57	Gpt4o-mini	0.11	0.09
					llama 3.1 8b	0.15	0.14
					llama 3.1 70b	0.12	**0.21**
					Gemini 1.5	**0.17**	0.18
Column Ambig.	$4.07	$9.44	253	88	Gpt4o-mini	0.19	0.14
					llama 3.1 8b	0.10	0.14
					llama 3.1 70b	0.15	0.19
					Gemini 1.5	**0.21**	**0.21**
Total	$5.15	$25.67	319	164	**Average**	0.12	0.21

Table 10: Comparison of SQUAB-generated and All-LLM-generated ambiguous tests for Beaver tables.

How Effectively Do LLMs Address Ambiguous Queries on Enterprise Data? Table 10 compares SQUAB and All-LLM on the Beaver dataset. By limiting hallucination effects and reducing the number of not executable queries, SQUAB generates twice the All-LLM valid tests at a fifth of the cost. Comparing the results with the tests on the Ambrosia tables (Table 9), Table 10 also reveals a drop in performance for all LLMs. For *Scope* and *Column ambiguity*, such reduction leads to a change in the best-performing LLM (Gemini 1.5 outperforms LLama-70b). This underscores that real-world performance differs significantly from public results.

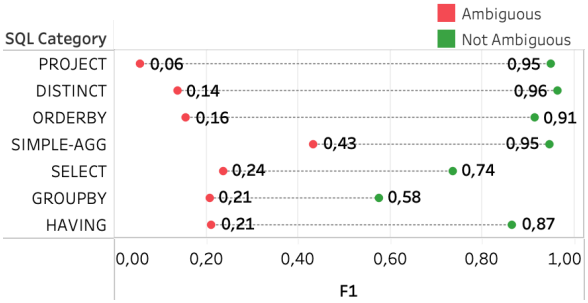

Figure 2: Comparison between ambiguous queries and not across categories. Results for Gemini 1.5 Pro on Beaver tables.

To isolate the impact of ambiguity on Beaver, we compared ambiguous vs. unambiguous *Column ambiguity* tests using Gemini 1.5 Pro (Figure 2). While unambiguous queries consistently score higher, ambiguity causes significant F1-score drops across SQL categories, most dramatically for PROJECT queries (0.95 to 0.06). This confirms ambiguity is a primary factor in performance degradation on enterprise data, underscoring SQUAB's value for benchmarking.

How do models perform on unanswerable questions? SQUAB offers significant advantages over All-LLM in scalability, generating a larger number of tests, with higher level of complexity, at a fraction of the cost. As shown in Tables 11 and 12, SQUAB consistently produces more tests (e.g., 513 vs. 458 for Beaver) while maintaining significantly lower costs (e.g., $7.67 vs. $39.08 for Beaver). SQUAB-generated tests are more challenging, as reflected by lower accuracy scores: All-

Test Type	$Cost SQUAB	$Cost all LLM	#tests SQUAB	#tests all LLM	Models	Accuracy SQUAB	Accuracy all LLM
Ambrosia							
Miss. Col.	$1.07	$0.76	213	162	Gpt4o-mini	**0.66**	**0.92**
					llama3.1 8b	0.00	0.01
					llama3.1 70b	0.29	0.76
					Gemini1.5 pro	0.55	0.90
Calc. Unans.	$0.88	$0.68	160	148	Gpt4o-mini	**0.43**	**1.00**
					llama3.1 8b	0.00	0.00
					llama3.1 70b	0.07	0.31
					Gemini1.5 pro	0.29	0.99
Out of Scope	$0.45	$0.75	123	144	Gpt4o-mini	**0.91**	**0.99**
					llama3.1 8b	0.00	0.04
					llama3.1 70b	0.35	0.38
					Gemini1.5 pro	0.90	0.99
Total	$2.40	$2.19	496	454	**Average**	0.37	0.61

Table 11: Comparison between SQUAB and All-LLM tests for Ambrosia tables.

Test Type	$Cost SQUAB	$Cost all LLM	#tests SQUAB	#tests all LLM	Models	Accuracy SQUAB	Accuracy all LLM
Beaver							
Miss. Col.	$3.30	$13.26	222	164	Gpt4o-mini	**0.41**	**0.87**
					llama3.1 8b	0.00	0.01
					llama3.1 70b	0.00	0.00
					Gemini1.5 pro	0.28	0.75
Calc. Unans.	$1.67	$12.99	89	159	Gpt4o-mini	**0.45**	**0.99**
					llama3.1 8b	0.00	0.00
					llama3.1 70b	0.00	0.00
					Gemini1.5 pro	0.24	0.94
Out of Scope	$2.70	$12.83	202	135	Gpt4o-mini	**0.78**	**0.99**
					llama3.1 8b	0.00	0.00
					llama3.1 70b	0.00	0.02
					Gemini1.5 pro	0.71	0.97
Total	$7.67	$39.08	513	458	**Average**	0.24	0.46

Table 12: Comparison between SQUAB and All-LLM tests generated for Beaver tables.

LLM achieves an average accuracy of 0.61 on Ambrosia tables, compared to 0.37 for SQUAB.

Error patterns differ by test category. For *Missing Column* tests, models often attempt to predict the missing entity, leading to invalid queries. For *Calculation Unanswerable* and *Out-of-Scope* tests, models frequently attempt to infer answers despite the unanswerable nature of the queries. Of these, *Out-of-Scope* tests are the least challenging, as models easily recognize when a question exceeds the SQL context. The most impacted model family is Llama 3.1, especially in enterprise settings.

How Effective is the Test Generation? To assess SQUAB's current effectiveness in an enterprise context, we conducted an annotation study with a company. One domain expert evaluated the quality of SQUAB'g generated tests across three of their enterprise tables. We used three metrics: (i) *Naturalness*: how natural and human-like the generated question appears. (ii) *Alignment*: whether the query accurately translates the question's intent. (iii) *Correctness*: whether the test adheres to its category definition. Each metric was evaluated on a three-level scale: *Low*, *Medium*, and *High*. We uniformly sampled eight tests per category, yielding 24 tests. The annotation study (reported in Appendix B) demonstrates that SQUAB produces high-quality tests across ambiguity and unanswerable categories, with 88% of all evaluations for both types falling into the *High* category.

Is SQUAB Extensible to New Ambiguity Types? To assess the effort required to integrate a new ambiguity type into SQUAB, we simulate this process. We select an ambiguity not currently covered: the *type-token ambiguity*, where a term can refer to a general category (type) or a specific instance (token), e.g., in "I paid for the same car," "car" could mean the same model or the exact same physical vehicle (Li et al., 2024b). One author, acting as a user extending the framework, implemented this new ambiguity type. The process, from defining the ambiguity and designing examples to coding the necessary pattern identification, metadata, and test generation logic took roughly 3 hours and resulted in less than 100 lines of Python code. This experiment demonstrates SQUAB's design facilitates extension for new ambiguity patterns. Details are given in Appendix C. Note that SQUAB is readily extensible to *multi-table* patterns. In this study, SQUAB deliberately focuses on single-table patterns to isolate the model's ability to handle ambiguity/unanswerability from the confounding effects of join complexity; extending the framework with *multi-table* patterns is left to future work.

5 Related Work

Converting NL questions to formal queries is challenged by ambiguity and unanswerability (Mu et al., 2024; Chen et al, 2021; Stengel-Eskin et al., 2024). Several Semantic Parsing benchmarks address this (Table 2). NoisySP (Wang et al., 2023) and Pythia (Veltri et al., 2023) generate ambiguous questions via table modifications (WikiSQL (Zhong et al., 2017), WTQ (Shi et al., 2020)) or templates. Damber (Papicchio et al., 2024) also uses templates for *Column ambiguity*. Am-

biQT (Bhaskar et al., 2023) uses LLMs to create broader ambiguities (e.g., joins, aggregates) on Spider (Yu et al., 2018), while Ambrosia (Saparina and Lapata, 2024) adds scope and attachment types. ArcherFish (Floratou et al., 2024) focuses on domain-specific customization. Unanswerability has also been explored in conversational AI (Dong et al., 2024; Yu et al., 2019), distinct from static SP. Most prior benchmarks use predefined templates or manual crafting, limiting ambiguity coverage. Crucially, unlike SQUAB and apart from Damber, they often do not differentiate proprietary from general-purpose data, despite known performance gaps (Papicchio et al., 2023; Chen et al., 2024).

6 Conclusion and Future directions

We presented SQUAB, an automatic framework for generating ambiguous and unanswerable queries for Semantic Parsing (Text2SQL). Our experiments show that SQUAB reduces test-generation costs by up to 99% compared with human-crafted tests, while better capturing real-world question complexity than a GPT-4o baseline. SQUAB enables rigorous robustness evaluation of LLMs and highlights the need for dynamic, tailored benchmarks that reflect enterprise use cases.

Future work could extend SQUAB beyond SP to fact-checking (Nakov et al., 2021; Guo et al., 2022), data analysis (He et al., 2024), and knowledge-graph querying (Feng et al., 2024). Another direction involves analyzing enterprise query workloads (Agrawal et al., 2000) to derive tests capturing domain-specific ambiguities and unanswerability. Given the encouraging results of synthetic training data for NLP tasks (Bussotti et al., 2024), we also envision positioning SQUAB as a generator of training data for SP models that need large, high-quality annotated datasets to be effective (Muennighoff et al., 2025; Papicchio et al., 2025).

Limitations

While SQUAB demonstrates significant advantages in generating challenging and cost-effective tests for semantic parsing, the current work has several limitations:

Scope of Task and Language Coverage: SQUAB's current implementation and evaluation are focused on semantic parsing (NL-to-SQL) and primarily on English language queries. Its direct applicability and performance on other tasks, such as fact-checking or general data analysis, or its efficacy for generating tests in languages other than English, have not yet been assessed. Extending the framework to these areas would require further research and adaptation of the generation pipeline.

Coverage of Ambiguity Types and Extensibility Framework: SQUAB currently ships with implementations for six specific categories of ambiguity and unanswerability. While our experiments demonstrate that the framework can be extended to new ambiguity types (such as type-token ambiguity) with modest effort, the generation of tests for any ambiguity type not yet implemented requires user-driven development. This involves defining the ambiguity's characteristics, designing the logic for pattern identification and test generation within the pipeline, and coding these components. Systematically discovering and automatically implementing entirely novel or highly nuanced domain-specific ambiguities from enterprise query workloads, without prior definition by a user, remains beyond the current automated capabilities.

Out-of-Scope UDF Verification: For the "Out-Of-Scope" unanswerable category, while SQUAB generates questions involving User-Defined Functions (UDFs) intended to be outside SQL's scope (e.g., forecasting), the current system does not programmatically verify that a generated UDF is definitively non-existent or inexpressible in standard SQL. This verification is currently a manual assessment or deferred.

Test Generation Process and User Input: SQUAB automates test generation based on an input database schema and content. However, the current pipeline is largely autonomous post-initiation. More interactive mechanisms, allowing users to guide or refine the test generation process actively, or to provide feedback on intermediate generations, are not yet implemented.

Complexity of Generated SQL: While SQUAB aims to generate challenging tests, the complexity of the SQL queries is guided by the templates used for each category. The system does not currently have a dynamic mechanism to control or progressively increase SQL complexity beyond these initial template designs.

Generation Cost Estimates: AMBROSIA lacks per-question costs and effort varies by ambiguity type, we estimate per-item cost as total annotation time divided by the number of ambiguous questions, applying this mean to the subset in Table 9.

Acknowledgements

This study was carried out within the FAIR - Future Artificial Intelligence Research and received funding from the European Union Next-GenerationEU (PIANO NAZIONALE DI RIPRESA E RESILIENZA (PNRR) – MISSIONE 4 COMPONENTE 2, INVESTIMENTO 1.3 – D.D. 1555 11/10/2022, PE00000013). This manuscript reflects only the authors' views and opinions, neither the European Union nor the European Commission can be considered responsible for them. This work has also been supported by the French government, through the 3IA Côte d'Azur Investments in the IA-cluster project managed by the National Research Agency (ANR) with the reference number ANR-23-IACL-0001.

References

Sanjay Agrawal, Surajit Chaudhuri, and Vivek R Narasayya. 2000. Automated selection of materialized views and indexes in sql databases. In *VLDB*, volume 2000, pages 496–505.

Adithya Bhaskar, Tushar Tomar, Ashutosh Sathe, and Sunita Sarawagi. 2023. Benchmarking and improving text-to-sql generation under ambiguity. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7053–7074.

Jean-Flavien Bussotti, Luca Ragazzi, Giacomo Frisoni, Gianluca Moro, and Paolo Papotti. 2024. Unknown claims: Generation of fact-checking training examples from unstructured and structured data. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pages 12105–12122. Association for Computational Linguistics.

Peter Baile Chen, Fabian Wenz, Yi Zhang, Moe Kayali, Nesime Tatbul, Michael Cafarella, Çağatay Demiralp, and Michael Stonebraker. 2024. Beaver: an enterprise benchmark for text-to-sql. *arXiv preprint arXiv:2409.02038*.

Mark Chen et al. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Mingwen Dong, Nischal Ashok Kumar, Yiqun Hu, Anuj Chauhan, Chung-Wei Hang, Shuaichen Chang, Lin Pan, Wuwei Lan, Henghui Zhu, Jiarong Jiang, Patrick Ng, and Zhiguo Wang. 2024. Practiq: A practical conversational text-to-sql dataset with ambiguous and unanswerable queries. *Preprint*, arXiv:2410.11076.

Yanlin Feng, Simone Papicchio, and Sajjadur Rahman. 2024. Cypherbench: Towards precise retrieval over full-scale modern knowledge graphs in the llm era. In *Annual Meeting of the Association for Computational Linguistics*.

Avrilia Floratou, Fotis Psallidas, Fuheng Zhao, Shaleen Deep, Gunther Hagleither, Wangda Tan, Joyce Cahoon, Rana Alotaibi, Jordan Henkel, Abhik Singla, Alex Van Grootel, Brandon Chow, Kai Deng, Katherine Lin, Marcos Campos, K. Venkatesh Emani, Vivek Pandit, Victor Shnayder, Wenjing Wang, and Carlo Curino. 2024. Nl2sql is a solved problem... not! In *Conference on Innovative Data Systems Research*.

Zhijiang Guo, Michael Schlichtkrull, and Andreas Vlachos. 2022. A survey on automated fact-checking. *Transactions of the Association for Computational Linguistics*, 10:178–206.

Xinyi He, Mengyu Zhou, Xinrun Xu, Xiaojun Ma, Rui Ding, Lun Du, Yan Gao, Ran Jia, Xu Chen, Shi Han, and 1 others. 2024. Text2analysis: A benchmark of table question answering with advanced data analysis and unclear queries. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 18206–18215.

Katalin É Kiss. 2006. Quantifier scope ambiguities. *The Blackwell companion to syntax*, pages 1–34.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, and 1 others. 2024a. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36.

Margaret Y. Li, Alisa Liu, Zhaofeng Wu, and Noah A. Smith. 2024b. A taxonomy of ambiguity types for NLP. *CoRR*, abs/2403.14072.

Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605.

Christopher D. Manning and Hinrich Schütze. 2001. *Foundations of statistical natural language processing*. MIT Press.

Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. 2024. Clarifygpt: A framework for enhancing llm-based code generation via requirements clarification. *Proc. ACM Softw. Eng.*, 1(FSE).

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. 2025. s1: Simple test-time scaling. *Preprint*, arXiv:2501.19393.

Preslav Nakov, David P. A. Corney, Maram Hasanain, Firoj Alam, Tamer Elsayed, Alberto Barrón-Cedeño, Paolo Papotti, Shaden Shaar, and Giovanni Da San Martino. 2021. Automated fact-checking for assisting human fact-checkers. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal,*

Canada, 19-27 August 2021, pages 4551–4558. ijcai.org.

Simone Papicchio, Paolo Papotti, and Luca Cagliero. 2023. Qatch: Benchmarking sql-centric tasks with table representation learning models on your data. In *Neural Information Processing Systems*.

Simone Papicchio, Paolo Papotti, and Luca Cagliero. 2024. Evaluating ambiguous questions in semantic parsing. In *2024 IEEE 40th International Conference on Data Engineering Workshops (ICDEW)*, pages 338–342. IEEE.

Simone Papicchio, Simone Rossi, Luca Cagliero, and Paolo Papotti. 2025. Think2sql: Reinforce llm reasoning capabilities for text2sql. *Preprint*, arXiv:2504.15077.

Philip Resnik. 1993. Semantic classes and syntactic ambiguity. In *Human Language Technology: Proceedings of a Workshop Held at Plainsboro, New Jersey, March 21-24, 1993*.

Irina Saparina and Mirella Lapata. 2024. Ambrosia: A benchmark for parsing ambiguous questions into database queries. In *Neural Information Processing Systems*.

Tianze Shi, Chen Zhao, Jordan Boyd-Graber, Hal Daumé III, and Lillian Lee. 2020. On the potential of lexico-logical alignments for semantic parsing to SQL queries. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1849–1864, Online. Association for Computational Linguistics.

Elias Stengel-Eskin, Kyle Rawlins, and Benjamin Van Durme. 2024. Zero and few-shot semantic parsing with ambiguous inputs. In *The Twelfth International Conference on Learning Representations*.

Enzo Veltri, Gilbert Badaro, Mohammed Saeed, and Paolo Papotti. 2023. Data ambiguity profiling for the generation of training examples. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 450–463. IEEE.

Bin 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 *Annual Meeting of the Association for Computational Linguistics*.

Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. 2020. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *Advances in neural information processing systems*, 33:5776–5788.

Tao Yu, Rui Zhang, Heyang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, and 5 others. 2019. CoSQL: A conversational text-to-SQL challenge towards cross-domain natural language interfaces to databases. 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)*, pages 1962–1979, Hong Kong, China. Association for Computational Linguistics.

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. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *EMNLP*, pages 3911–3921. ACL.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv 1709.00103*.

Appendices

A Main dataset characteristics

We evaluate SQUAB on two benchmarks: Ambrosia (Saparina and Lapata, 2024) and Beaver(Chen et al., 2024). Ambrosia is a synthetic benchmark specifically designed to capture various forms of ambiguity in natural language queries over relational databases. In contrast, Beaver is a large-scale enterprise database containing 99 real-world tables. To manage inference costs while maintaining statistical robustness, we uniformly sample 33% of Beaver's tables for our evaluation. See Table 13 for a summary of the database statistics.

For Ambrosia, we leverage all available tests for vagueness and attachment, restricting to single-table settings except for scope ambiguity, which in Ambrosia is present only with multi-table patterns. In this case, we construct a denormalized variant of Ambrosia's schema to ensure compatibility with our generation pipeline. For unanswerable query evaluation, we select the top 33 Ambrosia tables sorted by decreasing arity to align with Beaver's complexity.

B SQUAB with enterprise data

To assess SQUAB's current effectiveness in a practical enterprise context, we conducted an annotation study with a company. One domain expert evaluated the quality of SQUAB generated tests across three of their enterprise tables. We used three metrics: (i) *Naturalness*: how natural and human-like the generated question appears. (ii)

	#Databases	#Tables	#Attributes	#Tuples
Ambrosia				
Vague	71	5.57	4.72	3.57
Attach.	97	4.54	4.44	4.36
Scope	243	1.74	7.12	7.43
Enterprise				
Beaver	1	33	12.09	9.47×10^3

Table 13: Database statistics. Scope refers to the denormalized version. Beaver refers to the sampled version.

Alignment: whether the query accurately translates the question's intent. (iii) *Correctness*: whether the test adheres to its category definition. Each metric was evaluated on a three-level scale: *Low*, *Medium*, and *High*. To streamline the process, we uniformly sampled eight tests per category, yielding 24 annotated tests. However, for *Scope* and *Attachment* ambiguities, SQUAB generated fewer than eight tests per table due to their imposed constraints in table patterns.

The results in Table 14 of the annotation study demonstrates that SQUAB produces high-quality tests across ambiguity and unanswerable categories, with 88% of all evaluations for both test types falling into the *High* category, showcasing SQUAB's strong ability to generate natural, aligned, and category-correct tests.

C Type-Token Ambiguity

This section describes the new ambiguity type as a study of the usability of SQUAB. The *type-token* ambiguity arises when a term in a natural language question can plausibly refer either to a general category (type) or to individual instances (tokens) of that category (Li et al., 2024b). For example, in the query "How many cars did we sell in 2024?", the term "cars" may refer to the number of individual vehicles sold (token-level interpretation, typically mapped to `COUNT(*)` and the predicted query), or to the number of distinct car models sold (type-level interpretation, mapped to `COUNT(DISTINCT Model)`. Disambiguating such queries is challenging, especially when schema context is limited or when models lack fine-grained understanding of semantic roles.

In Table 15, we illustrate the generation process for the *type-token* ambiguity. The first step, *Pattern Identification*, detects columns that represent categories or types used to classify records. In this example, the *CarSales* table contains a column

Test Category	Metrics	*Low*	*Medium*	*High*	# Tests
Ambiguity					
Scope	Naturalness	-	1	13	14
	Alignment	-	2	12	
	Correctness	-	-	14	
Attach	Naturalness	1	2	13	16
	Alignment	-	-	16	
	Correctness	-	-	16	
Colum Ambig.	Naturalness	-	4	20	24
	Alignment	3	6	15	
	Correctness	-	2	22	
% Total		2%	10%	88%	54
Unanswerable					
Missing Col.	Naturalness	-	3	21	24
	Alignment	-	-	24	
	Correctness	-	-	24	
Calc. Unans.	Naturalness	3	3	18	24
	Alignment	-	-	24	
	Correctness	9	-	15	
OOS Unans.	Naturalness	-	6	18	24
	Alignment	-	3	21	
	Correctness	-	-	24	
% Total		5%	7%	88%	72

Table 14: Manual Annotations

Type-Token
Table: CarSales(CarId, Model, SellDate)
PI: Replacement Attribute: Model
RM: Ambiguous Term: Car,
SQL Interpretations:
`- SELECT COUNT(*)FROM CarSales WHERE SellDate = 2024`
`- SELECT COUNT(DISTINCT(Model))FROM CarSales WHERE SellDate = 2024`
Question: How many cars did we sell in 2024?

Table 15: Generation process for *Type-Token ambiguity*.

named *Model*, which denotes the type of car. In the second step, the *Relational Metadata Generator* produces the ambiguous term *car*, which can semantically refer either to a specific instance (token) or a general category (type). Finally, a natural language question — "How many cars did we sell in 2024?" — is generated based on SQL templates reflecting both interpretations. Each step in the pipeline is executed via a single LLM call.

The overall crafting process took approximately three hours, including the design of ambiguity patterns and implementation. The most challenging component was the *Pattern Identification* step, where the engineer must define a schema pattern capable of producing a type-token ambiguous term. However, since SQUAB is designed modularly, new patterns can be easily integrated or existing ones

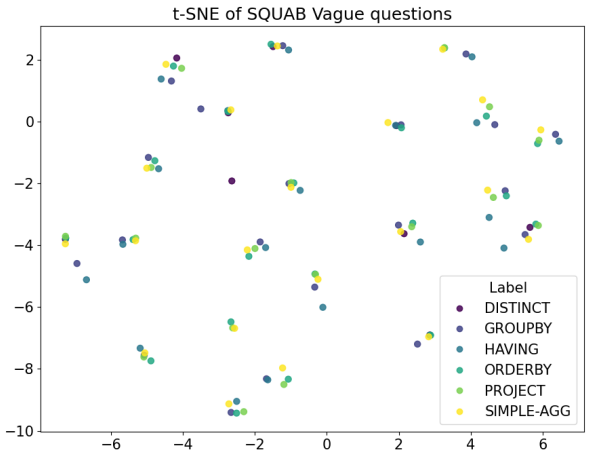

Figure 3: The variability of the generated questions for different SQL tags is highlighted in various colors. Visible clusters are formed based on questions generated from the same database.

reused, facilitating rapid development and extensibility.

D Variability of synthetically generated questions

In this section, we conduct a small study to assess the variability of questions generated by SQUAB. We focus on the *Column Ambiguity* subset from the Ambrosia database and analyze how questions differ across SQL clause tags. Each question is encoded with *all-MiniLM-L6-v2*[5], a compact fine-tuned variant of *MiniLM-L12-H384* (Wang et al., 2020).

Fig. 3 visualizes the resulting embeddings using t-SNE (Maaten and Hinton, 2008), with points colored by SQL tag. Questions sharing the same tag do not form coherent clusters; instead, clusters align with the source database used to construct the questions. This indicates that SQUAB generated questions are highly variable with respect to SQL tags—if tag semantics dominated, clusters would form by tag rather than by database.

E Prompting Details

This section provides the detailed prompts used in both the SQUAB and all-LLM settings for generating ambiguous and unanswerable questions.

E.1 SQUAB prompting

This section contains the prompts used in the SQUAB setting for both ambiguity and unanswerable test generation. SQUAB is composed of three main steps: *Pattern Identification*, *Relational Metadata Generation*, and *Test Generation*.

We use a single base prompt for the *Test Generation* one for ambiguity generation Fig. 6 and one for unanswerable generation Fig. 7. The placeholders `{ambig_definition}`, `{ambig_example}`, `{queries}`, `{metadata}`, and `{database}` are replaced with the ambiguity definition, an example of the ambiguity, the SQL queries that answer the question following the ambiguity rules, the metadata generated in the previous step, and the database content, respectively. The prompt for unanswerable generation is similar, with the only difference being that it contains the unanswerability definition and examples instead of ambiguity ones. For the relational metadata generation, we use a different prompt for each ambiguity (Fig. 8 and Fig. 9) and unanswerable category (Fig. 10, Fig. 11, and Fig. 12). As an example, Fig. 8 shows the prompt used to generate the ambiguous term for the *Column Ambiguity* category.

E.2 All-LLM prompting

This section contains the system prompts used in the all-LLM setting for both ambiguity (Fig. 5) and unanswerable (Fig. 4) test generation. The placeholders `{definition}` and `{examples}` are replaced with the unanswerability/ambiguity definition and generation examples, respectively. The user prompt is the same for both ambiguity and unanswerable generation, and contains the table schema.

[5] https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

Your task is to generate a list of question-answer pairs for the given table based on the unanswerability definition.
The answer is an unanswerable query that match the question.
The query is unanswerable because it cannot be answered from the given database (e.g., requires information or functionality outside standard SQL
Ambiguity Definition
{definition}
Question–Answer pair examples
{examples}
Output format
Given the table as input, generate a list of question–answer pairs in JSON format as follows:

```
[
  {
    "nl_question": "the ambiguous question
        that follows the definition",
    "target": ["the list of different SQL
        interpretations of the ambiguous
        question"]
  }
]
```

Figure 4: All-LLM synthetic data generation system prompt for unanswerable questions. The placeholders {definition} and {examples} are replaced with the unanswerability definition and generation examples, respectively.

Your task is to generate a list of question-answer pairs for the given table based on the ambiguity definition.
Ambiguous Definition
{definition}
Question-Answers pairs examples
{examples}
Output format
Given the table as input, generate a list of question–answer pairs in JSON format as follows:

```
[
  {
    "nl_question": "the ambiguous question
        that follows the definition",
    "target": ["the list of different SQL
        interpretations of the ambiguous
        question"]
  }
]
```

Figure 5: All-LLM synthetic data generation system prompt for ambiguous questions. The placeholders {definition} and {examples} are replaced with the ambiguity definition and generation examples, respectively.

You are a helpful assistant who writes a natural language (NL) question. You are provided with a definition of ambiguity, the SQL queries that answer the question following the ambiguity rules, and a database containing the answers. You may also receive metadata helping you in generating the question. Your task is to write the NL question following these guidelines:
- All unformatted table and column names must be replaced with plain words, preferably synonyms.
- Make the question as short as possible, but do not miss any part of the question like order-by (e.g., remove unnecessary words or paraphrase). Yet, you must check the relevant tables to ensure that the question and its interpretations express the same request as the queries and would yield the same answer. Example: You can modify ”fitness training program” into ”training program” and omit the unnecessary word “fitness” only if ”training program” cannot be confused with other columns in different tables.
- You must maintain ambiguity when writing the question and reading each interpretation.
- If the projected column name can be inferred, remove it from the final output
Output Format
Provide the answer in JSON format as follows

```json
{
    ”question”: ”the generated question”,
}
```

Ambiguity Definition
ambig_definition
Ambiguity Example
ambig_example
queries
queries
Metadata
metadata
Database
database

Figure 6: SQUAB synthetic data generation prompt for ambiguous questions. In blue there are the placeholders substituted based on each test category generation.

You are a helpful assistant who writes a natural language (NL) question from SQL query.
You are provided with the SQL query that answers the question, a database where to run the query, and some metadata.
Your task is to write the NL question following these guidelines:
- All unformatted table and column names must be replaced with plain words, preferably synonyms.
- Make the question as short as possible (e.g., remove unnecessary words or paraphrase). Still, you must check the relevant tables to ensure that the question is the same request as the query and will yield the same answer. Example: You can modify ”fitness training program” into ”training program” and omit the unnecessary word “fitness” only if ”training program” cannot be confused with other columns in different tables.
- If the projected column name can be inferred, remove it from the final output
Output Format
Provide the answer in JSON format as follows

```json
{
    ”question”: ”the generated question”,
}
```

Examples
examples
queries
queries
Metadata
metadata
Database
database

Figure 7: SQUAB synthetic data generation prompt for unanswerable questions. In blue there are the placeholders substituted based on each test category generation.

Generate an ambiguous label for a group of column names, given the table name and database name, such that the label can substitute all the names in the group in natural language questions. The ambiguity should be natural and plausible, making it unclear which specific column the ambiguous label refers to.
Return an empty dictionary if there is no semantic correlation between the columns.
Steps
1. **Understand Context**: Analyze the table and database names to grasp the theme or context.
2. **Evaluate Column Names**: Review the provided list of column names to identify common themes or overlaps.
3. **Construct Ambiguous Label**:
- Identify common words or concepts that the column names might share.
- Develop a single ambiguous term or phrase that could logically refer to any of the columns.
- Ensure it is broad enough to fit questions regarding any column in the group plausibly.
Output Format
Provide a list of ambiguous labels, such as a single phrase or a few words. Do not include additional explanations, and keep the format concise.
Examples
Input:

Num to generate: 2
Database Name: ”UniversityRecords”,
Table Name: ”StudentPerformance”,
Columns: [”MathScore”, ”PhysicsScore”, ”BiologyScore”]
Output:
[”subject score”, ”grade”]

Output:

[”personal identifier”]

Notes
- Ensure that the ambiguous label remains a plausible term that might be used in everyday queries or conversations about the topic.
- Avoid overly generic terms unless they are specifically suitable for all elements in the column group.

Figure 8: SQUAB relational metadata prompt for *Column Ambiguity* test generation.

Identify the semantic relationship between two provided names and determine if one is an Entity and the other is a Component. Note that a component can also be an element present in the entities.
Steps
1. Analyze the first name to determine if it can be categorized as an Entity or a Component.
2. Analyze the second name to determine if it can be categorized as a Component or an Entity.
3. Evaluate if the selected component is a meaningful part or attribute of the selected entity.
Output Format
Return the answer as JSON enclosed in “‘json “‘ with two keys: entity and component.

“‘json
{
”entity”: ”the name that represents the entity”,
”component”: ”the name that represents the component.”
}
“‘

Examples
Example 1:

– Input: ”Engine”, ”Car”
– Output:

“‘json
{
”entity”: ”Car”,
”component”: ”Engine”
}
“‘

Example 2:

– Input: ”Brand name”, ”Store name”
– Output:

“‘json
{
”entity”: ”Store name”,
”component”: ”Brand name”
}
“‘

Example 3:

– Input: ”Hospital”, ”Amenities”
– Output:
“‘json
{
”entity”: ”Hospital”,
”component”: ”Amenities”
}
“‘

Figure 9: SQUAB relational metadata prompt for *Scope* test generation.

Generate suggestions for new columns to add to a database table, including the type of column (categorical or numerical) and sample data for each column based on the given table name, database name, and existing column names.
Steps
1. **Analyze Provided Information**: Review the table name, database name, and existing column names to determine the context and purpose of the table.
2. **Infer Potential Data Gaps**: Consider common or useful additional columns that could complement or enhance the data in the table.
3. **Suggest New Columns**:
- Determine if each suggested column should be categorical or numerical based on the inferred data gap.
- Provide a rationale for why each new column would be a beneficial addition.
4. **Generate Sample Data**: For each suggested column, provide sample data that fits the column type.
Output Format
Provide the output in a structured JSON format:

```
“‘json
{”suggested_columns”: [
    {
        ”column_name”: ”[suggested_column_name]”,
        ”column_type”: ”[categorical/numerical]”,
        ”description”: ”the description of the column”,
        ”sample_data”: [”[sample_value1]”, ”[sample_value2]”, ...]
    }, ... ]}
“‘
```

Ensure the suggestions are relevant to the context implied by the existing column names.
Examples
Input

```
Num to generate: 2
Table Name: Customers
Database Name: SalesDB
Table Schema: [”customer_id”, ”name”, ”email”, ”purchase_history”]
```

Output

```
“‘json
{”suggested_columns”: [
    {”column_name”: ”customer_segment”,
    ”column_type”: ”categorical”,
    ”description”: ”The customer segment for the sales”,
    ”sample_data”: [”Regular”, ”VIP”, ”New”]},
    {”column_name”: ”average_spending”,
    ”column_type”: ”numerical”,
    ”description”: ”the average spending of the customer”,
    ”sample_data”: [100.0, 250.5, 300.3]}
]}
“‘
```

Notes
- Consider the context provided by the existing columns to ensure the suggestions add value.
- For databases associated with specific industries (e.g., finance, healthcare, retail), leverage common industry practices for enhancing data tables.

Figure 10: SQUAB relational metadata prompt for *Missing Column* test generation.

Create a User-Defined Function (UDF) executable in SQL using the given table schema.
The output should be structured in JSON format. The UDF must not be as obvious as the percentage.
The UDF name must be different from existing columns to avoid any confusion with column names and should
not contain any overlapping names or prefixes with the column names.
The semantic of the UDF must be different from the semantic of each column.
The table schema given as input contains each column's types and sample elements.
The "udf_name" contains the call of the user-defined function with the column names separated by commas.
Steps
1. **Analyze the Table Schema**: Understand the provided table schema.
2. **Design the UDF**: Create a hypothesis for the function using some of the columns available in the schema.
3. **Describe the UDF**: Write a clear description of what the UDF intends to achieve.
Output Format
The output should be a JSON object with the following structure:
- **udf_name**: A descriptive and relevant name for the User-Defined Function with the called columns. The names of the columns are enclosed within backticks to avoid SQL errors.
- **udf_description**: A detailed explanation of the function's intended operations.
- **udf_output_type**: the output data type of the UDF. It can be "categorical" or "numerical".
The output must also contain the Python code that executes the logic of the UDF. The python code is enclosed in "'python "' after the JSON.
Generate at most the num of examples given as input, each separated by "# New UDF"
Example:

```
# New UDF
"'json
{
    "udf_name": "calculate_interest_rate(`Age`, `Income`, `Credit_score`)",
    "udf_description": "This UDF attempts to calculate a score based on the 'age', 'income', and '
        credit_score' columns.",
    "udf_output_type": "numerical"
}
"'
"'python
def calculate_interest_rate(account_id, customer_id, balance, credit_score, loan_history):
    interest_rate = (balance * 0.05) + (credit_score * 0.02) – (loan_history * 0.01)
    return interest_rate
"'
```

Notes
- Ensure the python syntax is precise and executable for valid hypothetical values.
- As input you will also get the number of UDF to generate

Figure 11: SQUAB relational metadata prompt for *Calculation Unanswerable* test generation.

Create a User-Defined Function (UDF) that is executable but unanswerable using only the specified table schema.
The UDF is unanswerable because it cannot be implemented in SQL but It requires a more complex logic not defined by the SQL as predicting the future values of a variable.
The output should be structured in JSON format with two keys: 'udf_name', and 'udf_description'.
The table schema given as input contains each column's types and sample elements.
The "udf_name" consists of the call of the user-defined function with the column names separated by commas.
Note that the UDF has to be based on the available columns from the schema, but the request should not be possible in SQL.
Generate at most the num of examples given as input.
Steps
1. **Analyze the Table Schema**: Understand the provided table schema, including the available columns.
2. **Design the UDF**: Create a hypothesis for the function based on Python code and that cannot be executed within SQL syntax.
3. **Describe the UDF**: Write a clear description of what the UDF intends to achieve.
Output Format
The output should be a JSON object containing a list of "suggested_udfs" with the following structure:
- **udf_name**: A descriptive and relevant name for the User-Defined Function with the called columns. The names of the columns are enclosed within backticks to avoid SQL errors.
- **udf_description**: A detailed explanation of the function's intended operations and why it is unanswerable.
- **udf_output_type**: the output data type of the UDF. It can be "categorical" or "numerical".
Example:
Provide the output in a structured JSON format:

```
```json
{
"suggested_udfs": [
    {
        "udf_name": "predict_interest_rate(`Age`, `Income`, `Credit_score`)",
        "udf_description": "This UDF attempts to predict the interest rate based on Age, Income, and
            credit score."
        "udf_output_type": "numerical"
    },
    ...
    ]
}
```
```

Notes
- Remember, the goal is to ensure the UDF is based on existing columns but logically requires a different execution that is not available in SQL.
- As input you will also get the number of UDF to generate

Figure 12: SQUAB relational metadata prompt for *Out-Of-Scope* test generation.