BEAVER: An Enterprise Benchmark for Text-to-SQL

Peter Baile Chen[1] **Fabian Wenz**[1,3] **Yi Zhang**[2] **Devin Yang**[1] **Justin Choi**[1]
Nesime Tatbul[1] **Michael Cafarella**[1] **Çağatay Demiralp**[1,2] **Michael Stonebraker**[1]
[1]MIT [2]AWS AI Labs [3]TUM
Correspondence: peterbc@mit.edu

Abstract

Existing text-to-SQL benchmarks have largely been constructed from web tables with human-generated question-SQL pairs. LLMs typically show strong results on these benchmarks, leading to a belief that LLMs are effective at text-to-SQL tasks. However, how these results transfer to enterprise settings is unclear because tables in enterprise databases might differ substantially from web tables in structure and content. To address this gap, we introduce a new dataset BEAVER, the first *private enterprise* text-to-SQL benchmark. This dataset includes natural language queries and SQL statements collected from real query logs. Experimental results show that *off-the-shelf* LLMs struggle with this dataset. We identify three main reasons for the poor performance: (1) enterprise table schemas are more intricate than those found in public datasets, making SQL generation inherently more challenging; (2) business-oriented queries tend to be more complex, often involving multi-table joins, aggregations, and nested queries; (3) public LLMs cannot train on private enterprise data warehouses that are not publicly accessible, and therefore it is difficult for the model to learn to solve (1) and (2). We believe BEAVER will facilitate future research in building text-to-SQL systems that perform better in enterprise settings.

1 Introduction

LLMs have shown potential for solving text-to-SQL tasks on existing datasets, such as Spider, KaggleDBQA, and Bird (Li et al., 2024; Sen et al., 2019; Yu et al., 2018; Lee et al., 2021). For example, on Spider, GPT-4 can achieve an execution accuracy above 85% (Gao et al., 2024). However, these datasets focus on public tables and question-SQL pairs written by crowd-sourced annotators. As such, they do not represent real-world enterprise settings for the following reasons.

First, enterprise databases, typically designed for internal business use, often utilize more intricate schemas than tables from public datasets. Hence, understanding them may require database or business-specific knowledge. Public LLMs are mainly trained on public data. In contrast, enterprise data is private, which makes public LLMs lack access to such knowledge. Recent work (Kandpal et al., 2023) has shown that LLMs do not perform well on data domains they have never seen before. Consequently, public LLMs may not perform well on enterprise text-to-SQL tasks.

Second, questions posed to enterprise databases are generally more complex than questions from public datasets. Public datasets are usually small and typically general-purpose. Questions from these datasets are often collected from annotators who are not enterprise users, database admins, or business analysts from specific data domains. For instance, the Spider dataset (Yu et al., 2018) was annotated by 11 computer science undergraduates. Therefore, the questions posed tend to be simple and may only involve one or two tables. In contrast, queries on enterprise databases typically involve joins and aggregates over multiple tables.

Third, enterprise databases often contain a large number of tables, rows, and columns. The scale of enterprise tables makes selecting the relevant tables for text-to-SQL even more challenging. These size issues are often absent from the public databases used for benchmarking text-to-SQL.

To study the above issues, we have curated a dataset, called BEAVER, derived by anonymizing three real-world private data warehouses. SQL statements were gathered from actual user query logs and reports, and corresponding natural language questions were formulated in collaboration with experienced database administrators. Specifically, we benchmarked recent *off-the-shelf* LLMs on BEAVER. These models achieved around 5% end-to-end execution accuracy, demonstrating the challenging nature of our dataset. This illustrates that *off-the-shelf* LLMs trained on public datasets

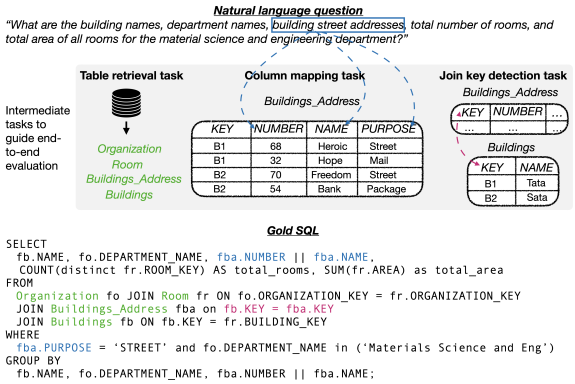

Figure 1: End-to-end text-to-SQL task and intermediate tasks.

are unable to generalize to the same text-to-SQL tasks when presented with real data warehouse data.

In summary, our contributions are as follows: (1) We introduce BEAVER, the first *private enterprise* text-to-SQL benchmark, for benchmarking text-to-SQL models under enterprise settings. This dataset includes tables from private and real enterprise data warehouses, annotated question-SQL pairs, and annotations of column mapping, gold tables, and join keys for each question. Existing LLM-powered text-to-SQL systems are not trained on them. (2) We evaluated LLM-based text-to-SQL approaches on BEAVER and show their dramatically degraded performance. Additionally, we introduced three intermediate tasks that can serve as better indicators of end-to-end performance and highlight potential areas for improvement, underscoring the value of our benchmark for evaluation. (3) We provide an extensive error analysis that reveals why enterprise data and queries are challenging for LLMs. We then propose steps to address these challenges, informing future text-to-SQL systems that can perform better on enterprise data and queries.

2 Dataset

As described in Section 1, existing public datasets do not reflect the high schema and query complexity found in enterprise data warehouses. To address this gap, we collected datasets from three enterprise data warehouses and annotated them with real-world question-SQL pairs. We describe the end-to-end text-to-SQL task and suggest intermediate tasks that can guide the end-to-end evaluation. Then, we provide details on the datasets and annotations.

2.1 Task Formulation

Following the standard problem setup of text-to-SQL, the input to an LLM includes a natural language question and a database of tables, and the output is a SQL statement whose execution answers the question. A database includes a set of tables. Each table includes a schema (that describes the names of and data types of each column) and instances of each table column. Besides the end-to-end text-to-SQL task, we defined three *intermediate* tasks that can aid in solving it, as shown in Figure 1. While addressing the end-to-end task does not strictly require solving these intermediate tasks, they serve as valuable indicators of final performance and provide insights for potential improvements.

Intermediate Task 1: Table Retrieval. As discussed in Section 1, enterprise databases often consist of a vast number of tables and columns, posing a challenge in fitting all relevant information into an LLM's input prompt. To address this, models must retrieve a relevant subset of connected tables (Chen et al., 2024) from the entire database to generate an accurate SQL statement. For instance, to answer the user question shown in Figure 1, models need to identify four relevant tables from a significantly larger database. The input for this task includes a natural language question and the database, while the output is a set of relevant tables that can help construct the correct SQL statement.

Intermediate Task 2: Column Mapping. To generate a correct SQL statement from a natural language question, models need to identify information referenced in the user question and determine which table columns contain this information. As illustrated in Figure 1, models need to recognize that retrieving details about "*building street address*" requires accessing columns `NUMBER`, `NAME`, and `PURPOSE` in table `Building_Address`. The input for this task consists of a natural language question, a set of tables, and a list of topic phrases (e.g., "*building street address*"), while the output is a mapping from each input topic phrase to the corresponding table columns. As discussed in Section 1, the complex schema of enterprise databases makes this task more challenging than with public datasets. Beyond mapping user-mentioned information to table columns, mapping information to specific table instances is also valuable. For example, the phrase "*material science and engineering*

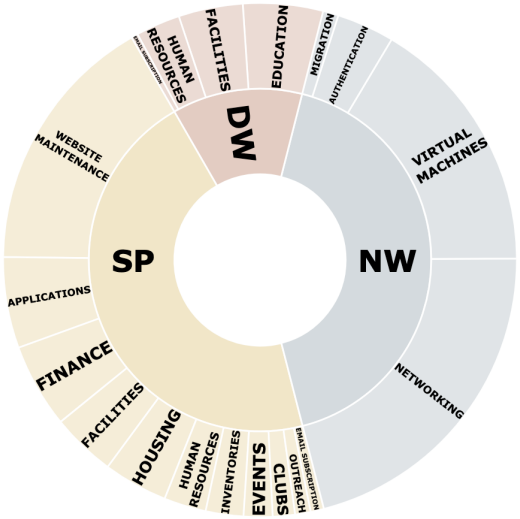

Figure 2: Domain distribution of our data warehouses.

department" corresponds to instance `'Materials Science and Eng'` in column `DEPARTMENT_NAME` in table `Organization`. However, we do not include instance mapping as an intermediate task due to the significant complexity of annotations (detailed in Appendix A), which complicates the quantitative evaluation of this task.

Intermediate Task 3: Join Key Detection. Finally, models must also understand how to link multiple tables by identifying overlapping or semantically related columns (i.e., join keys). As illustrated in Figure 1, connecting information between tables `Buildings_Address` and `Buildings` requires recognizing that column `KEY` serves as the join key. The input for this task consists of a natural language question and a set of relevant tables, while the output is the list of join keys necessary to establish the connections between them.

2.2 Annotation

Databases. We collected table information directly from three private data warehouses, including column names, column types, rows, and join keys (if exist). The first data warehouse, called DW, consists of 97 tables and 1530 columns from an existing Oracle data warehouse. The second, NW, comprises 366 tables and 2708 columns spanning five MySQL databases. The third, SP, contains 349 tables and 2717 columns from two MySQL databases. The domain distribution of the three data warehouses is summarized in Figure 2 (details can be found in Appendix B).

SQL statements. To reflect the true complexity of queries posed on enterprise databases, we first collected real user query logs and reports from source organizations. We then extracted SQL statements from these real logs and reports.

Natural language questions. Four graduate students and two professional database administrators from the data warehouse support group collectively constructed natural language questions for the collected SQL statements. The students first collaboratively generated the natural language question for the corresponding SQL statement. Then, they passed these questions to the two database administrators for review of quality.

Intermediate tasks. Gold answers for all intermediate tasks were annotated collectively by students and database administrators, leveraging existing SQL parsers and tools. Gold tables required by the SQL statement and join keys in the gold SQL statement were first extracted using existing SQL parsing packages[1], and then manually checked. Regarding column mapping, for each topic phrase (e.g., *building names*) mentioned in a user question, mappings to relevant table columns were annotated as a pair of (topic phrase, columns names). A full annotation for an example question can be found in Appendix C.

2.3 Statistics

Table 1 summarizes the dataset size, table statistics, and query complexity of our dataset as well as two popular datasets on public tables: Spider (Yu et al., 2018) and Bird (Li et al., 2024). Similar to (Lan et al., 2023; Li et al., 2024), we measure query complexity along three dimensions: the average number of joins per query, which indicates the number of tables that need to be joined to include sufficient information to answer the user question; the average number of aggregations per query, which indicates the number of aggregation keywords such as `max`, `count`, `group by` that appear in a SQL statement; and the nesting depth which indicates how deep sub-queries appear (e.g., `SELECT ... FROM (SELECT ...)` has a nesting depth of two). Compared to all existing datasets, BEAVER has the largest number of tables per database and the highest query complexity. Figure 3 visualizes the query complexity across all datasets and complexity dimensions.

[1]Packages used are sqlglot and sqlparse.

Table 1: Dataset size, table statistics, and query statistics of text-to-SQL datasets.

Dataset	#Queries	#DB	Avg. #Table/DB	Avg. #Cols/Table	Avg. #Joins/query	Avg. #Aggregation/query	Avg. Nesting depth/query
Spider (Dev)	1034	20	4.05	5.44	0.506	0.854	1.09
Bird (Dev)	1534	11	6.82	10.6	0.918	0.663	1.09
BEAVER	261	8	102.0	8.57	3.77	2.49	1.98

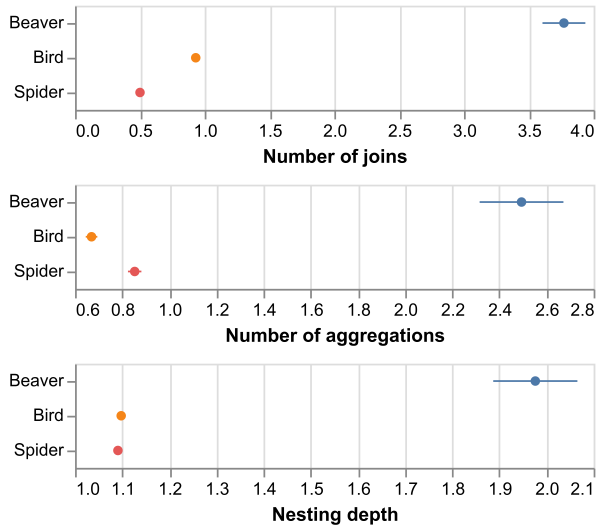

Figure 3: The mean values for the number of joins, aggregations, and nesting depth for Spider, Bird, and BEAVER.

3 Benchmark

In this section, we outline the benchmarking setup and evaluation metrics for the end-to-end and intermediate tasks. We then evaluate recent retrievers and LLMs on BEAVER and existing public text-to-SQL datasets. We further analyze the results in relation to the characteristics of enterprise databases discussed in Section 1, aiming to identify directions for improvement.

3.1 Experimental setup

Datasets. We evaluated our dataset separately on each database. However, doing so for Spider and Bird make them too simple compared with our dataset. As seen in Table 1, the average number of tables per database is significantly smaller on Spider and Bird compared to BEAVER. Therefore, we aggregated tables from all databases to construct a centralized database, resulting in 81 tables for Spider and 75 for Bird. This step ensures the table corpus sizes of Spider and Bird are comparable with our dataset (102.0 tables per database). For Spider and Bird, we still track the original databases of each table to evaluate SQL statements.

Retrieval-augmented Generation (RAG).
Table Retrieval. As seen in Table 1, BEAVER contains a large number of tables per database and a large number of columns per table, which makes it challenging to fit all this information into LLM's input prompt. A common method to enhance LLMs with knowledge from a large external data source is retrieval-augmented generation (Lewis et al., 2020). Following this approach, instead of directly providing LLMs with the user question and the full database schema for SQL generation, a dense retriever first selects the top-k tables based on the semantic relatedness (cosine similarity between embeddings) between the user question and the table schema[2]. In particular, we considered three recent embedding models for dense retrieval: UAE-Large-V1 (Li and Li, 2023), Stella_en_400M_v5[3], and GTE-large-en-v1.5 (Li et al., 2023).
SQL Generation. An SQL statement is generated given a user question and schema of the top-k most relevant tables. Additionally, we can provide the gold annotations for the intermediate tasks defined in Section 2.1 to test models' ability to generate SQL statements when provided with more hints. We adopted 1-shot prompting and benchmarked on multiple recent LLMs, including GPT-4o (Achiam et al., 2023), Llama-3.1-Instruct (70B and 8B) (Touvron et al., 2023), and Qwen-2.5-Instruct (72B and Coder-32B) (Yang et al., 2024; Team, 2024). Temperature (a random seed) was set to 0 to minimize randomness. Detailed 1-shot prompts for SQL generation can be found in Appendix E.1.

Intermediate tasks. Intermediate tasks include table retrieval, column mapping, and join key detection. The experimental setup for table retrieval has already been described above. For column mapping and join key detection, we follow the task definitions outlined in Section 2.1. Specifically, we provide LLMs with natural language questions and the corresponding gold tables. Additionally, for column mapping, we provide the topic phrases from the gold column mappings rather than having

[2]Table schema are serialized as space-separated strings of table names and column names.

[3]https://huggingface.co/dunzhang/stella_en_400M_v5

LLMs generate them, ensuring a controlled and reproducible evaluation. Detailed 1-shot prompts for these tasks can be found in Appendix E.2. Since the Spider and Bird datasets do not include gold annotations for these intermediate tasks, we randomly sampled 50 queries from each dataset and manually annotated the gold column mappings and join keys for comparison.

3.2 Evaluation metrics

Table retrieval. To evaluate table retrieval performance, we use the standard metrics of precision, recall, and F1 of the retrieved tables compared to gold tables. However, these metrics may be insufficient. Since a SQL statement is unlikely to be correctly generated without retrieving *all* gold tables, we also introduce perfect recall (PR), which measures the percentage of questions with all gold tables retrieved.

SQL generation. Following (Yu et al., 2018; Li et al., 2024), we use execution accuracy to evaluate the end-to-end performance. Execution accuracy is defined as 1 if the predicted and gold SQL statements produce the same results and 0 otherwise.

Column mapping and join key detection. To evaluate the predicted and gold answers for these two intermediate tasks, we use two metrics: exact match and F1 score. The exact match score is 1 if the predicted answer perfectly matches the gold answer and 0 otherwise. However, since this criterion may be too strict, we also incorporate the F1 score, which is computed based on the precision and recall of matched units. For column mapping, each (topic phrase, column names) pair is treated as a basic comparison unit, while for join key detection, each join key serves as the basic comparison unit.

3.3 Overall performance

Table retrieval performance. Table 2 shows the performance on the intermediate task of table retrieval. We see that recall and perfect recall @ top-k on BEAVER are the lowest across all models and datasets. On average, recall @ top-10 is 44.9 points lower on BEAVER compared to Spider and 42.8 points lower than Bird across all retriever models. Similarly, perfect recall @ top-10 is, on average, 82.0 points lower than Spider and 77.2 points lower than Bird. These results highlight the significant challenge of accurately identifying the relevant set of tables needed to answer a user query in the context of an enterprise database.

Table 2: Table retrieval performance of all embedding models. PR is the percentage of questions with all gold tables retrieved.

	Top-5				Top-10			
	P	R	F1	PR	P	R	F1	PR
UAE-Large-V1								
Spider	29.1	96.4	43.5	94.6	14.9	98.6	25.4	97.9
Bird	34.8	91.3	49.1	82.6	18.9	97.5	31.1	94.5
BEAVER	29.7	38.0	32.2	7.7	21.2	52.8	29.4	14.6
Stella_en_400M_v5								
Spider	30.1	99.6	44.9	99.3	15.1	100	25.8	99.9
Bird	35.4	93.0	50.0	85.6	18.9	97.8	31.2	95.1
BEAVER	35.2	44.3	37.9	10.3	23.9	58.6	32.8	20.3
GTE-large-en-v1.5								
Spider	29.0	96.6	43.3	94.1	14.9	99.0	25.5	98.1
Bird	33.2	87.8	47.0	76.7	18.5	96.0	30.5	91.7
BEAVER	29.2	36.7	31.3	9.2	21.0	51.5	28.9	14.9

Table 3: 1-shot column mapping performance. Results are sampled on 50 queries from each dataset.

	Spider		Bird		BEAVER	
	F1	Exact	F1	Exact	F1	Exact
GPT-4o	80.8	64.0	76.9	52.0	64.0	17.9
Qwen2.5-72B-It	79.8	62.0	85.8	70.0	68.9	18.5
Qwen2.5-32B-It	67.8	52.0	65.8	44.0	65.0	12.1
Llama3.1-70B-It	75.8	60.0	70.5	44.0	58.9	10.4
Llama3.1-8B-It	55.3	36.0	57.6	28.0	50.8	13.3

Column mapping performance. Table 3 shows the performance of models on the intermediate task of column mapping. On average, F1 on BEAVER is 10.4 points lower than Spider and 9.8 points lower than Bird across all models. Similarly, the average exact match score on BEAVER is 40.4 points lower than Spider and 33.2 points lower than Bird. These results highlight the difficulty of correctly identifying the relevant columns in BEAVER, making it significantly more challenging than both Spider and Bird. The low exact match performance (up to 19%) on BEAVER further suggests that while models can correctly map some keywords, they struggle to accurately map all relevant keywords within a given question.

Join key detection performance Table 4 presents the performance of models on the intermediate task of join key detection. On average, the exact match score on BEAVER is 23.8 points lower than Spider and 38.8 points lower than Bird. These results underscore the challenge of accurately identifying join keys in BEAVER, suggesting that this task is significantly more difficult compared to both Spider and Bird.

Table 4: 1-shot join key detection performance. Results are sampled on 50 queries from each dataset.

	Spider		Bird		BEAVER	
	F1	Exact	F1	Exact	F1	Exact
GPT-4o	62.7	58.8	88.3	80.0	76.4	46.7
Qwen2.5-72B-It	74.5	70.6	85.6	80.0	72.6	44.4
Qwen2.5-32B-It	70.6	70.6	81.0	75.0	66.4	38.5
Llama3.1-70B-It	51.0	47.1	89.5	82.5	68.6	38.5
Llama3.1-8B-It	60.8	52.9	63.3	57.5	44.5	13.0

Table 5: 1-shot end-to-end execution accuracy. Top-10 tables from the best-performing retriever model (*Stella_en_400M_v5*) were provided to the models.

	Spider	Bird	BEAVER
GPT-4o	69.5	30.9	3.83
Qwen2.5-72B-It	69.9	28.6	4.21
Qwen2.5-32B-It	71.7	27.7	1.92
Llama3.1-70B-It	60.3	25.8	1.15
Llama3.1-8B-It	51.1	13.8	0

End-to-end execution accuracy. As shown in Table 5, the end-to-end execution accuracy on BEAVER is the lowest across all datasets and models, with an average of 2.22 across all models, compared to 64.5 on Spider and 25.3 on Bird. This underscores the challenging nature of BEAVER. The low accuracy is likely influenced by poor performance on the intermediate tasks, as these tasks reflect models' abilities to handle different subcomponents of a SQL statement.

3.4 Analysis

As discussed in Section 1, BEAVER differs from public text-to-SQL datasets in three key aspects: (1) larger database size, (2) higher schema complexity, and (3) greater query complexity. In this section, we analyze how each of these factors impacts LLM performance. Additionally, we highlight the benefits of incorporating gold answers for the intermediate tasks, as summarized in Table 6.

Table 6: 1-shot execution accuracy on BEAVER when gold answers for intermediate tasks are provided, across different models. GT, M, and J refer to gold tables, column mappings, and join keys, respectively.

	GPT	Qwen2.5-It		Llama3.1-It	
	4o	72B	Coder-32B	70B	8B
Base	3.83	4.21	1.92	1.15	0.0
Base + GT	7.28	6.13	4.21	5.36	0.77
Base + GT + M	8.43	6.90	8.05	2.30	2.68
Base + GT + M + J	10.3	8.05	7.66	5.75	4.21

Table 7: Number of correctly answered questions over three buckets (0-4, 5-9, 9+) of each dimension of complexity.

Average number	0-4	5-9	9+
Join			
# total queries	192	47	14
# correct predictions	20	4	0
Aggregation			
# total queries	205	41	7
# correct predictions	16	8	0
Nesting depth			
# total queries	237	16	0
# correct predictions	22	2	0

Providing gold answers of each intermediate task increases performance. As discussed in Section 2.1, there are three intermediate tasks: table retrieval, column mapping, and join key detection. As shown in Table 6, averaging across all models, providing gold tables boosts execution accuracy by 113.8%. Incorporating gold column mappings further improves accuracy by 19.4%, and supplying gold join keys leads to an additional 26.8% increase. These results demonstrate that the quality of intermediate task outputs significantly impacts overall performance, making them valuable indicators for evaluation and key areas for improvement. Retriever models may retrieve tables that introduce noise (due to irrelevant tables) or lack essential information (due to missing gold tables). Consequently, the accuracy improvement when providing gold tables suggests that the large database size increases the task's difficulty. Furthermore, supplying gold column mappings further enhances performance, emphasizing that schema complexity presents a significant challenge for models in accurately performing column mapping.

Increased query complexity reduces performance. To examine the impact of query complexity on performance, Table 7 presents the number of correctly predicted SQL statements from GPT-4o when provided with gold tables, column mappings, and join keys, across different levels of query complexity (as defined in Section 2.3). The results indicate a clear trend: as query complexity increases, the number of correctly predicted SQL statements decreases. This demonstrates that higher query complexity negatively affects performance, making it more challenging to generate accurate SQL queries in our dataset.

Table 8: Common error types encountered in table retrieval and SQL generation tasks for retriever and LLM models, respectively.

Error types	% questions
Table retrieval (Stella_en_400M_v5)	
Not retrieving sufficient information	89.1
Misses connecting tables	6.52
Cannot handle domain-specific information	4.38
SQL generation (GPT-4o)	
Incorrect column mapping	59.1
Incorrect instance mapping	22.7
Unable to handle complex queries	27.3
Misses implicit assumptions	50.0

4 Error analysis

In the above, we provided an overview of the performance of *off-the-shelf* LLMs on BEAVER, indicating their limited capabilities of performing text-to-SQL in a real-world enterprise setting. Here, we discuss in detail the error sources during both table retrieval and SQL generation phases by examining randomly sampled 50 questions from our dataset. For table retrieval, we examined the performance of the best-performing retriever model (*Stella_en_400M_v5*). For SQL generation, we inspected the performance of the best-performing LLM (*GPT-4o*).

4.1 Table retrieval analysis

As seen in the top half of Table 8, the retriever model made three major mistakes during table retrieval. Firstly, *the retrieval model may not retrieve the set of tables with sufficient information to answer the user question*. For instance, given the user question "*What is the name of the building and fee of the shortest sessions?*" and a table corpus including the three tables shown in Figure 4, the retriever model retrieved table `SUBJECT_SESSION` to cover "shortest and longest sessions", and table `SUBJECT_DETAIL` to cover "fee". However, the model did not retrieve table `BUILDINGS` to cover "name of the building".

Secondly, *the retrieval model can miss connecting tables*. This occurs when models retrieved a set of tables that can cover information in the user question, but they might not be connected through join relationships, so other tables need to be used to connect these tables. For instance, given the user question "*What is the building name that accommodates the most students?*" and a table corpus including the three tables shown in Figure 5, the retriever model retrieved `FCLT_BUILDING` and `STUDENT_DIRECTORY` to cover "building name" and "students" respectively. However, these two tables can only be joined via `FCLT_ROOMS`, which was not retrieved. This shows that models are not necessarily aware of join relationships during retrieval, which leads to information not being connected.

Lastly, *retrieval models may not be able to retrieve correct tables if domain-specific information is involved*. For example, given the user question "*List the name of mailing lists, and name of the faculty who teaches in 2023 fall.*" that requires information from tables `MOIRA_LIST`, `MOIRA_LIST_DETAIL`, and `SUBJECT_OFFERED`, the retriever model only retrieved the table `SUBJECT_OFFERED`, but was unable to retrieve the other two tables that are related to "moira list", potentially because it does not know that "moira" is the name of the system used to manage mailing lists.

These behaviors suggest that existing retriever models struggle to retrieve relevant tables for a user question in the enterprise setting.

4.2 SQL generation analysis

As seen in the bottom half of Table 8, models made four major mistakes in SQL generation. Firstly, *models can map topics mentioned in user questions to incorrect columns (i.e., incorrect column mapping)*. For instance, given the user question "*What are the building names and building street addresses for the computer science department?*", GPT-4o mapped "*building street address*" to the column `BUILDING_ADDRESS`. However, GPT-4o is not aware that the same building can have multiple addresses for different purposes (e.g., street, mail, package), and thus failed to also map this topic to the column `ADDRESS_PURPOSE` and instance `'STREET'`. Column mapping also fails when user questions are vague. For instance, when network administrators pose questions like "*Provide information (including info on caches and security groups) for the virtual machine with ID [id].*", they would like to gather as much information as possible to perform diagnosis and monitoring. Therefore, the gold SQL statement is very comprehensive, whereas GPT-4o only predicted a few columns. The full example can be found in Appendix D.1.

Secondly, *models can map literals mentioned in user questions to incorrect instances (i.e., incorrect instance mapping)*. For example, given a user ques-

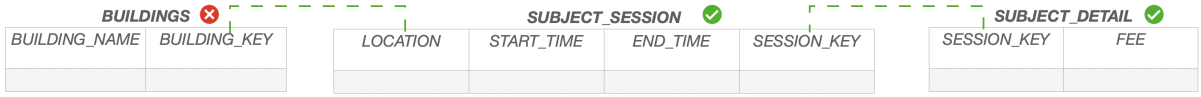

Figure 4: Schema of tables to illustrate retriever models did not retrieve sufficient information. A green tick means the table was retrieved, and a red cross means the table was not retrieved. Green dotted lines represent join relationships.

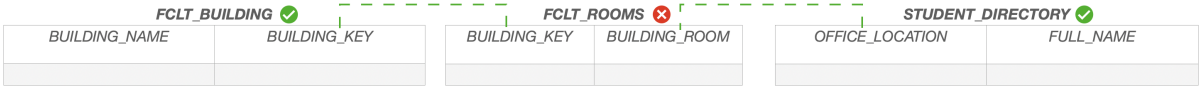

Figure 5: Schema of tables to illustrate retriever model did not retrieve connecting tables.

tion "*What is the total fee for all virtual sessions?*", GPT-4o associated the literal "virtual" with column `SESSION_LOCATION` and instance `'Virtual'`. While the column mapping is correct, the instance mapping is incorrect because `SESSION_LOCATION` includes multiple instances that represent virtual locations (e.g., `'webinar'`, `'remote'`, `'online via zoom'`), so the model would need to associate "virtual" with all these different instances or explore a more efficient filter for virtual locations.

Thirdly, models can fail to derive the correct SQL syntax when queries are complex. For instance, given the user question "For each course, list the cumulative number of courses held in the same year or preceding years.", the correct approach is to `partition` courses by year, sort courses by year, and restrict courses to those that have the same year or before using the function `range between unbounded preceding and current row`. However, GPT-4o was not able to use the window function in its prediction.

Finally, models cannot reflect implicit assumptions in SQL statements. For instance, when users pose questions like "*Provide information about virtual machines with ID [id].*", by default, they only want to know the information about *active* instances (i.e., not deleted). As such, the gold SQL statement includes the predicate `instances.deleted = 0`. However, GPT-4o was not able to recover this implicit assumption (and thus the predicate) in its SQL statement.

Overall, the error analysis highlights that retrieving relevant tables from a large corpus, performing schema mapping (both column mapping and instance mapping), and understanding complex queries are big challenges for models to solve enterprise-level text-to-SQL. Moreover, models might also need to deal with ambiguity and implicit assumptions in user questions.

5 Discussion

As discussed in Section 4, natural language questions posed by enterprise users may not explicitly specify all required information and can include implicit assumptions, although there exists an SQL statement that corresponds to the user question. As a result, the verbosity of questions varies depending on users' background knowledge, making schema mapping more challenging (as shown in Section 3.3) and lowering performance on the end-to-end task. In contrast, questions in public text-to-SQL datasets tend to be highly verbose, explicitly mentioning every column needed in the SQL statement. To address this challenge, we propose two possible solutions: (1) a human-in-the-loop iterative approach, where models generate clarifying questions and refine their outputs based on continuous human feedback, and (2) a comprehensive method that identifies all possible schema mappings and presents them to users for selection.

6 Conclusion

Text-to-SQL plays a crucial role in bridging natural language question answering with database querying. While off-the-shelf LLMs appear to perform well on existing text-to-SQL benchmarks, these benchmarks fail to capture the complexities of real-world enterprise environments. Unlike public datasets, enterprise settings involve domain-specific knowledge, a vast number of tables requiring an intermediate retrieval step, and increased schema and query complexity. Our findings demonstrate that enterprise queries pose significant challenges for off-the-shelf models, particularly in table retrieval and SQL generation. We aim for this work to lay the groundwork for future research on large-scale and complex text-to-SQL tasks.

7 Ethics

As mentioned in Section 2.2, we recruited four graduate students and two professional database administrators to perform the annotations. We ensure fair compensation for each person, considering the minimum salary of the region these volunteers are in. Because this dataset involves only factual annotations, no subjective opinions or personal information were collected, and thus, it should pose minimal risks to annotators and the general public. All database contents and questions will be anonymized according to rules set by the private organizations before releasing them to the public.

8 Limitations

Privacy and legal considerations restricted our access to private databases, limiting the diversity of domains represented in our dataset. Furthermore, in order to collect real SQL statements, we focused on query logs and reports. However, interpreting the intent of the SQL queries was difficult, making the generation of precise natural language questions a slow process. We plan to continue expand number of queries in our dataset in the future.

Acknowledgments

We gratefully acknowledge the support of DARPA ASKEM Award HR00112220042, the ARPA-H Biomedical Data Fabric project, a grant from Liberty Mutual, and the Croucher Scholarship.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Peter Baile Chen, Yi Zhang, and Dan Roth. 2024. Is table retrieval a solved problem? exploring join-aware multi-table retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2687–2699, Bangkok, Thailand. Association for Computational Linguistics.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-sql empowered by large language models: A benchmark evaluation. *Proc. VLDB Endow.*, 17(5):1132–1145.

Nikhil Kandpal, Haikang Deng, Adam Roberts, Eric Wallace, and Colin Raffel. 2023. Large language models struggle to learn long-tail knowledge. In *International Conference on Machine Learning*, pages 15696–15707. PMLR.

Wuwei Lan, Zhiguo Wang, Anuj Chauhan, Henghui Zhu, Alexander Li, Jiang Guo, Sheng Zhang, Chung-Wei Hang, Joseph Lilien, Yiqun Hu, Lin Pan, Mingwen Dong, Jun Wang, Jiarong Jiang, Stephen Ash, Vittorio Castelli, Patrick Ng, and Bing Xiang. 2023. Unite: A unified benchmark for text-to-sql evaluation. *Preprint*, arXiv:2305.16265.

Chia-Hsuan Lee, Oleksandr Polozov, and Matthew Richardson. 2021. KaggleDBQA: Realistic evaluation of text-to-SQL parsers. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2261–2273, Online. Association for Computational Linguistics.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2024. 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.

Xianming Li and Jing Li. 2023. Angle-optimized text embeddings. *arXiv preprint arXiv:2309.12871*.

Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023. Towards general text embeddings with multi-stage contrastive learning. *arXiv preprint arXiv:2308.03281*.

Jaydeep Sen, Fatma Ozcan, Abdul Quamar, Greg Stager, Ashish Mittal, Manasa Jammi, Chuan Lei, Diptikalyan Saha, and Karthik Sankaranarayanan. 2019. Natural language querying of complex business intelligence queries. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1997–2000.

Qwen Team. 2024. Qwen2.5: A party of foundation models.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin

Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zhihao Fan. 2024. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

A Complexity of instance mapping

Consider the user question "*List the long building names constructed before 1950 that have more than 100 employees and the built year and number of employees.*" which has a gold SQL statement of

```
SELECT * FROM (SELECT DISTINCT
    a.BUILDING_NAME_LONG, a.year_built,
    COUNT(distinct
    employee_directory.ID) OVER
    (PARTITION BY a.BUILDING_NAME_LONG,
    a.year_built) as num_employees
FROM (SELECT * FROM (SELECT DISTINCT
    FCLT_BUILDING_KEY,
    BUILDING_NAME_LONG, extract(year
    FROM TO_DATE(date_built,
    'MM/DD/YYYY')) as year_built FROM
    fclt_building_hist) WHERE
    year_built < 1950) a JOIN
    fclt_rooms ON
    fclt_rooms.FCLT_BUILDING_KEY =
    a.FCLT_BUILDING_KEY JOIN
    employee_directory ON
    employee_directory.OFFICE_LOCATION
    = fclt_rooms.BUILDING_ROOM ) WHERE
    num_employees > 100;
```

In this case, the literal "*100 employees*" should be mapped to

```
COUNT(distinct employee_directory.ID)
    OVER (PARTITION BY
    a.BUILDING_NAME_LONG, a.year_built)
    > 100
```

which involves one grouping and aggregation.

The literal "before 1950" should be mapped to

```
extract(year FROM TO_DATE(date_built,
    'MM/DD/YYYY'))  < 1950
```

which involves one custom function call.

As seen above, compared to column mappings, instance mapping is considerably more complex and much harder to evaluate. Therefore, instance mappings were not annotated.

B Domain distribution

Figure 6 include the detailed domain distribution of each data warehouse.

C Annotations

C.1 Full annotation of an example question

```
{
"question": "What is the current
    building key, building street
    address, city, state, and postal
    code of the history department?",
"sql": "SELECT DISTINCT
    d.FCLT_BUILDING_KEY,
    e.BUILDING_STREET_ADDRESS, d.CITY,
    d.STATE, d.POSTAL_CODE FROM
    FCLT_BUILDING_ADDRESS d JOIN
    FCLT_ROOMS a ON a.FCLT_BUILDING_KEY
    = d.FCLT_BUILDING_KEY JOIN
    FCLT_ORG_DLC_KEY b ON
    a.FCLT_ORGANIZATION_KEY =
    b.FCLT_ORGANIZATION_KEY JOIN
    MASTER_DEPT_HIERARCHY c ON
    b.DLC_KEY = c.DLC_KEY JOIN
    BUILDINGS e ON e.BUILDING_KEY =
    d.FCLT_BUILDING_KEY WHERE
    lower(c.DLC_NAME) =
    lower('History') AND
    d.ADDRESS_PURPOSE = 'STREET';",
"tables": [
  "FCLT_BUILDING_ADDRESS",
  "FCLT_ROOMS",
  "FCLT_ORG_DLC_KEY",
  "MASTER_DEPT_HIERARCHY",
  "BUILDINGS"
],
"mapping": {
  "building key": [
    "FCLT_BUILDING_ADDRESS.FCLT_BUILDING_KEY"
  ],
  "building street address": [
    "BUILDINGS.BUILDING_STREET_ADDRESS",
    "FCLT_BUILDING_ADDRESS.ADDRESS_PURPOSE"
  ],
  "city": [
    "FCLT_BUILDING_ADDRESS.CITY"
  ],
  "state": [
    "FCLT_BUILDING_ADDRESS.STATE"
  ],
  "postal code": [
    "FCLT_BUILDING_ADDRESS.POSTAL_CODE"
  ],
  "history department": [
    "MASTER_DEPT_HIERARCHY.DLC_NAME"
  ]
},
"join_keys": [
  [
    "FCLT_BUILDING_ADDRESS.FCLT_BUILDING_KEY",
    "FCLT_ROOMS.FCLT_BUILDING_KEY"
  ],
  [
    "FCLT_ROOMS.FCLT_ORGANIZATION_KEY",
    "FCLT_ORG_DLC_KEY.FCLT_ORGANIZATION_KEY"
  ],
  [
```

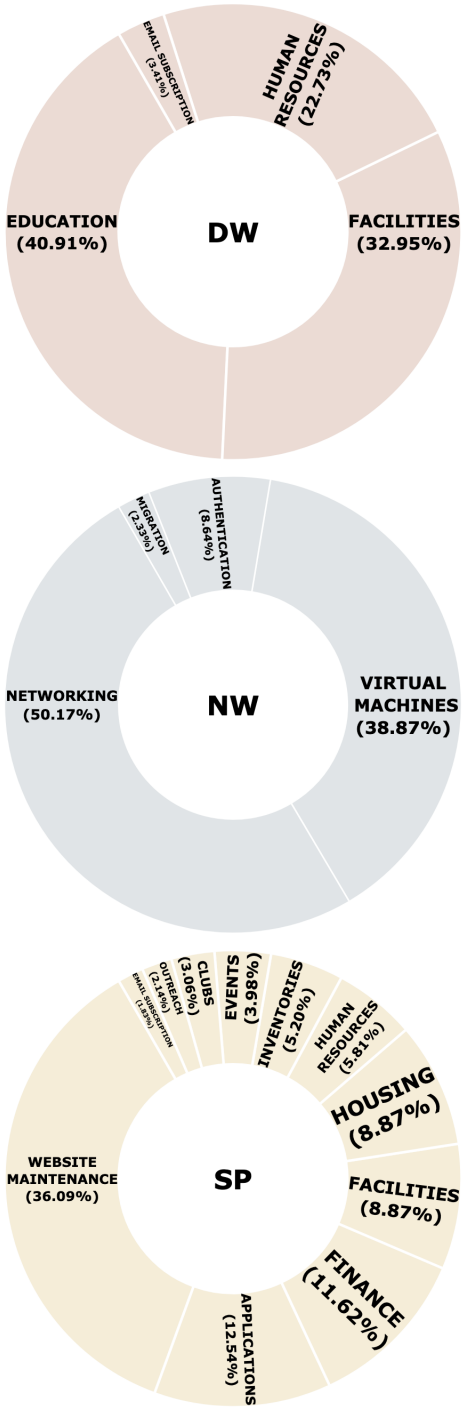

Figure 6: Detailed domain distribution of the source data warehouses.

```
    "FCLT_ORG_DLC_KEY.DLC_KEY",
    "MASTER_DEPT_HIERARCHY.DLC_KEY"
  ],
  [
    "FCLT_BUILDING_ADDRESS.FCLT_BUILDING_KEY",
    "BUILDINGS.BUILDING_KEY"
  ]
]
}
```

C.2 Annotation interface

We show the interface annotators used for converting SQL statements to natural language questions (Figure 7). Annotators receive the SQL statement and a set of potential natural language (NL) questions (created by GPT). They have the option to select one of the provided questions or compose their own.

D Examples for error analysis

D.1 Column mapping for vague questions

For the user question "*Provide information (including info caches, and security groups) for these VMs f5a08397-5aac-44b4-b359-f03ff6ce228a, e7c1acd1-6a47-4a08-8601-5022d4d50aa7.*", the gold and predicted SQL statements (by GPT-4o) are shown in Table 9. The predicted SQL statement only selects a few columns to return while the gold SQL statement includes a lot more information.

E Prompts

E.1 1-shot prompt for SQL generation

We use the 1-shot prompt in Table 10 for end-to-end SQL generation.

E.2 1-shot prompt for column mapping and join key detection

We use the 1-shot prompt in Table 11 for column mapping generation. We use the 1-shot prompt in Table 12 for join key generation.

Table 9: Gold and predicted SQL statement for the vague user question.

Gold SQL statement

```
SELECT instances.created_at AS instances_created_at, instances.updated_at AS
    instances_updated_at, instances.deleted_at AS instances_deleted_at,
    instances.deleted AS instances_deleted, instances.id AS instances_id,
    instances.user_id AS instances_user_id, instances.project_id AS
    instances_project_id, instances.image_ref AS instances_image_ref,
    instances.kernel_id AS instances_kernel_id, instances.ramdisk_id AS
    instances_ramdisk_id, instances.hostname AS instances_hostname,
    instances.launch_index AS instances_launch_index, instances.key_name AS
    instances_key_name, instances.key_data AS instances_key_data,
    instances.power_state AS instances_power_state, instances.vm_state AS
    instances_vm_state, instances.task_state AS instances_task_state,
    instances.memory_mb AS instances_memory_mb, instances.vcpus AS instances_vcpus,
(...33 columns omitted...)
instance_info_caches_1.created_at AS instance_info_caches_1_created_at,
    instance_info_caches_1.updated_at AS instance_info_caches_1_updated_at,
    instance_info_caches_1.deleted_at AS instance_info_caches_1_deleted_at,
    instance_info_caches_1.deleted AS instance_info_caches_1_deleted,
    instance_info_caches_1.id AS instance_info_caches_1_id,
    instance_info_caches_1.network_info AS instance_info_caches_1_network_info,
    instance_info_caches_1.instance_uuid AS instance_info_caches_1_instance_uuid,
    security_groups_1.created_at AS security_groups_1_created_at,
    security_groups_1.updated_at AS security_groups_1_updated_at,
    security_groups_1.deleted_at AS security_groups_1_deleted_at,
    security_groups_1.deleted AS security_groups_1_deleted, security_groups_1.id AS
    security_groups_1_id, security_groups_1.name AS security_groups_1_name,
    security_groups_1.description AS security_groups_1_description,
    security_groups_1.user_id AS security_groups_1_user_id,
    security_groups_1.project_id AS security_groups_1_project_id
FROM
instances LEFT OUTER JOIN instance_info_caches AS instance_info_caches_1 ON
    instance_info_caches_1.instance_uuid = instances.uuid LEFT OUTER JOIN
    (security_group_instance_association AS security_group_instance_association_1
    INNER JOIN security_groups AS security_groups_1 ON security_groups_1.id =
    security_group_instance_association_1.security_group_id AND
    security_group_instance_association_1.deleted = 0 AND security_groups_1.deleted
    = 0) ON security_group_instance_association_1.instance_uuid = instances.uuid
    AND instances.deleted = 0
WHERE instances.uuid IN ('f5a08397-5aac-44b4-b359-f03ff6ce228a',
    'e7c1acd1-6a47-4a08-8601-5022d4d50aa7')
```

Predicted SQL statement (GPT-4o)

```
SELECT instances.uuid AS instance_uuid, instances.display_name, instances.vm_state,
    instances.host, instance_info_caches.network_info, security_groups.name AS
    security_group_name, security_groups.description AS security_group_description
FROM
    instances
LEFT JOIN
    instance_info_caches ON instances.uuid = instance_info_caches.instance_uuid
LEFT JOIN
    security_group_instance_association ON instances.uuid =
        security_group_instance_association.instance_uuid
LEFT JOIN
    security_groups ON security_group_instance_association.security_group_id =
        security_groups.id
WHERE
    instances.uuid IN (
        'f5a08397-5aac-44b4-b359-f03ff6ce228a',
        'e7c1acd1-6a47-4a08-8601-5022d4d50aa7'
    )
```

Table 10: 1-shot prompt for SQL generation (content in parentheses is only available when gold column mappings or gold join keys are provided).

You are given a list of tables, a user question, (a mapping from information mentioned in the user question to columns in the provided tables, and join keys among the provided tables), your task is output a Oracle SQL statement that can be used to answer the user question based on the provided tables. You need to ensure that syntax and functions used in your SQL statement are appropriate for Oracle database. If you are unable to determine the SQL statement, output None. (You should use the provided mapping to determine which columns and tables should be used in the SQL statement. You should use the provided join keys to determine how to connect the tables in the SQL statement.)

```
CREATE TABLE fac_floor(
   WAREHOUSE_LOAD_DATE DATE,
   BUILDING_KEY VARCHAR2,
   FLOOR VARCHAR2,
   FLOOR_KEY VARCHAR2,
   EXT_GROSS_AREA NUMBER,
   ASSIGNABLE_AREA NUMBER,
   NON_ASSIGNABLE_AREA NUMBER,
   FLOOR_SORT_SEQUENCE VARCHAR2,
   LEVEL_ID VARCHAR2,
   BUILDING_WINGS_ID VARCHAR2,
   ACCESS_LEVEL VARCHAR2
)

CREATE TABLE fac_building(
   DATE_ACQUIRED VARCHAR2,
   DATE_OCCUPIED VARCHAR2,
   WAREHOUSE_LOAD_DATE DATE,
   NUM_OF_ROOMS NUMBER,
   FAC_BUILDING_KEY VARCHAR2,
   BUILDING_NUMBER VARCHAR2,
   PARENT_BUILDING_NUMBER VARCHAR2,
   PARENT_BUILDING_NAME VARCHAR2,
   PARENT_BUILDING_NAME_LONG VARCHAR2,
   BUILDING_NAME_LONG VARCHAR2,
   EXT_GROSS_AREA NUMBER,
   ASSIGNABLE_AREA NUMBER,
   NON_ASSIGNABLE_AREA NUMBER,
   SITE VARCHAR2,
   CAMPUS_SECTOR VARCHAR2,
   ACCESS_LEVEL_CODE NUMBER,
   ACCESS_LEVEL_NAME VARCHAR2,
   BUILDING_TYPE VARCHAR2,
   OWNERSHIP_TYPE VARCHAR2,
   BUILDING_USE VARCHAR2,
   OCCUPANCY_CLASS VARCHAR2,
   BUILDING_HEIGHT VARCHAR2,
   COST_CENTER_CODE VARCHAR2,
   COST_COLLECTOR_KEY VARCHAR2,
   LATITUDE_WGS NUMBER,
   LONGITUDE_WGS NUMBER,
   EASTING_X_SPCS NUMBER,
   NORTHING_Y_SPCS NUMBER,
   BUILDING_SORT VARCHAR2,
   BUILDING_NAMED_FOR VARCHAR2,
   BUILDING_NAME VARCHAR2,
   DATE_BUILT VARCHAR2
)
```

User question: List name and floor of the building with the largest floor number?
(Mapping: "name" in the user question refers to column `BUILDING_NAME` in table `fac_building` | "floor" in the user question refers to column `FLOOR` in table `fac_floor`)
(Join keys: Column `BUILDING_KEY` in table `FAC_FLOOR` joins with column `FAC_BUILDING_KEY` in table `FAC_BUILDING`)
SQL: SELECT DISTINCT B.BUILDING_NAME, A.FLOOR FROM FAC_FLOOR A JOIN FAC_BUILDING B ON A.BUILDING_KEY = B.FAC_BUILDING_KEY JOIN (SELECT max(f) as highest_floor FROM (SELECT CASE WHEN REGEXP_LIKE(FLOOR, '^\d+$') THEN TO_NUMBER(FLOOR) ELSE NULL END AS f FROM fac_floor)) ON (CASE WHEN REGEXP_LIKE(A.FLOOR, '^\d+$') THEN TO_NUMBER(FLOOR) ELSE NULL END) = highest_floor;

{tables}
User question: {user question}
(Mapping: {mapping})
(Join keys: {join keys})
SQL:

Table 11: 1-shot prompt for column mappings.

You are given a list of tables, a user question, and a list of keywords extracted from the user question. Your task is to map each keyword to the most relevant columns names from the given tables. The output should be a dictionary in JSON format, where each key is a keyword and the corresponding value is a list of relevant table column names. The final output need to be enclosed within <ans></ans> tags.

```
CREATE TABLE fac_floor(
   WAREHOUSE_LOAD_DATE DATE,
   BUILDING_KEY VARCHAR2,
   FLOOR VARCHAR2,
   FLOOR_KEY VARCHAR2,
   EXT_GROSS_AREA NUMBER,
   ASSIGNABLE_AREA NUMBER,
   NON_ASSIGNABLE_AREA NUMBER,
   FLOOR_SORT_SEQUENCE VARCHAR2,
   LEVEL_ID VARCHAR2,
   BUILDING_WINGS_ID VARCHAR2,
   ACCESS_LEVEL VARCHAR2
)

CREATE TABLE fac_building(
   DATE_ACQUIRED VARCHAR2,
   DATE_OCCUPIED VARCHAR2,
   WAREHOUSE_LOAD_DATE DATE,
   NUM_OF_ROOMS NUMBER,
   FAC_BUILDING_KEY VARCHAR2,
   BUILDING_NUMBER VARCHAR2,
   PARENT_BUILDING_NUMBER VARCHAR2,
   PARENT_BUILDING_NAME VARCHAR2,
   PARENT_BUILDING_NAME_LONG VARCHAR2,
   BUILDING_NAME_LONG VARCHAR2,
   EXT_GROSS_AREA NUMBER,
   ASSIGNABLE_AREA NUMBER,
   NON_ASSIGNABLE_AREA NUMBER,
   SITE VARCHAR2,
   CAMPUS_SECTOR VARCHAR2,
   ACCESS_LEVEL_CODE NUMBER,
   ACCESS_LEVEL_NAME VARCHAR2,
   BUILDING_TYPE VARCHAR2,
   OWNERSHIP_TYPE VARCHAR2,
   BUILDING_USE VARCHAR2,
   OCCUPANCY_CLASS VARCHAR2,
   BUILDING_HEIGHT VARCHAR2,
   COST_CENTER_CODE VARCHAR2,
   COST_COLLECTOR_KEY VARCHAR2,
   LATITUDE_WGS NUMBER,
   LONGITUDE_WGS NUMBER,
   EASTING_X_SPCS NUMBER,
   NORTHING_Y_SPCS NUMBER,
   BUILDING_SORT VARCHAR2,
   BUILDING_NAMED_FOR VARCHAR2,
   BUILDING_NAME VARCHAR2,
   DATE_BUILT VARCHAR2
)

User question: List name and floor of the building with the largest floor number?
Keywords: name, floor
Mapping:<ans>{“name”: [“fac_building.BUILDING_NAME”], “floor”: [“fac_floor.FLOOR”]}</ans>

{tables}
User question: {user question}
Keywords: {keywords}
Mapping:
```

Table 12: 1-shot prompt for join key detection.

You are given a list of tables, and a user question. Your task is to identify which columns from different tables can be used for joining. Each valid pair of joinable columns should be represented as a list. The output should be a list of lists in JSON format, enclosed within <ans></ans> tags.

```
CREATE TABLE fac_floor(
   WAREHOUSE_LOAD_DATE DATE,
   BUILDING_KEY VARCHAR2,
   FLOOR VARCHAR2,
   FLOOR_KEY VARCHAR2,
   EXT_GROSS_AREA NUMBER,
   ASSIGNABLE_AREA NUMBER,
   NON_ASSIGNABLE_AREA NUMBER,
   FLOOR_SORT_SEQUENCE VARCHAR2,
   LEVEL_ID VARCHAR2,
   BUILDING_WINGS_ID VARCHAR2,
   ACCESS_LEVEL VARCHAR2
)

CREATE TABLE fac_building(
   DATE_ACQUIRED VARCHAR2,
   DATE_OCCUPIED VARCHAR2,
   WAREHOUSE_LOAD_DATE DATE,
   NUM_OF_ROOMS NUMBER,
   FAC_BUILDING_KEY VARCHAR2,
   BUILDING_NUMBER VARCHAR2,
   PARENT_BUILDING_NUMBER VARCHAR2,
   PARENT_BUILDING_NAME VARCHAR2,
   PARENT_BUILDING_NAME_LONG VARCHAR2,
   BUILDING_NAME_LONG VARCHAR2,
   EXT_GROSS_AREA NUMBER,
   ASSIGNABLE_AREA NUMBER,
   NON_ASSIGNABLE_AREA NUMBER,
   SITE VARCHAR2,
   CAMPUS_SECTOR VARCHAR2,
   ACCESS_LEVEL_CODE NUMBER,
   ACCESS_LEVEL_NAME VARCHAR2,
   BUILDING_TYPE VARCHAR2,
   OWNERSHIP_TYPE VARCHAR2,
   BUILDING_USE VARCHAR2,
   OCCUPANCY_CLASS VARCHAR2,
   BUILDING_HEIGHT VARCHAR2,
   COST_CENTER_CODE VARCHAR2,
   COST_COLLECTOR_KEY VARCHAR2,
   LATITUDE_WGS NUMBER,
   LONGITUDE_WGS NUMBER,
   EASTING_X_SPCS NUMBER,
   NORTHING_Y_SPCS NUMBER,
   BUILDING_SORT VARCHAR2,
   BUILDING_NAMED_FOR VARCHAR2,
   BUILDING_NAME VARCHAR2,
   DATE_BUILT VARCHAR2
)

User question: List name and floor of the building with the largest floor number?
Join keys: <ans>[["FAC_FLOOR.BUILDING_KEY", "FAC_BUILDING.FAC_BUILDING_KEY"]]</ans>

{tables}
User question: {user question}
Join keys:
```

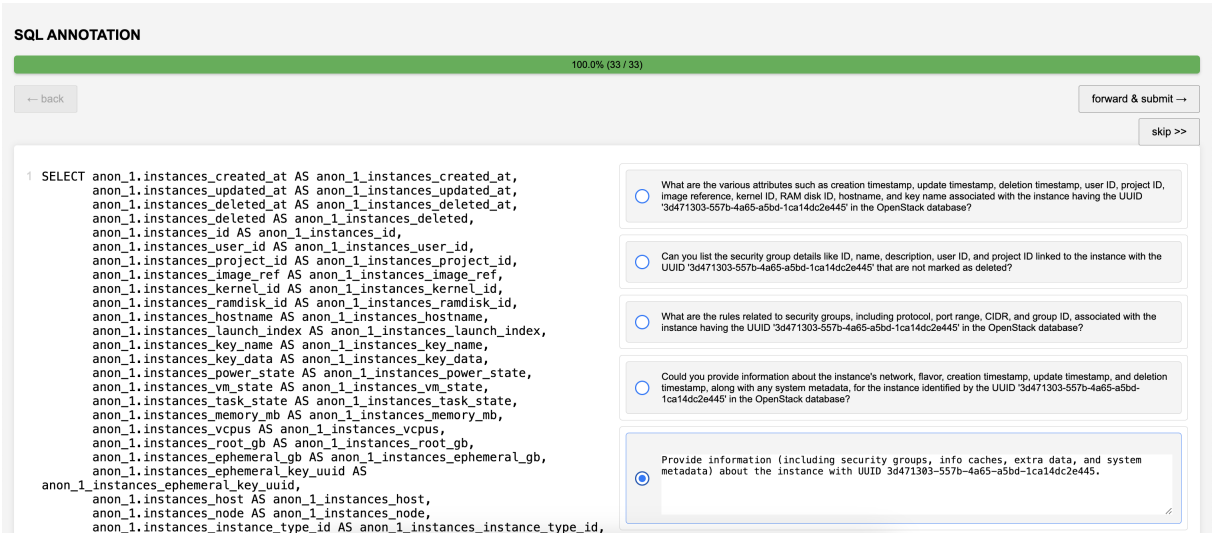

Figure 7: Interface for annotating natural language questions.