Fine-tuning Small LLMs with Graph-Modeled Schemas for Multi-table NL2SQL

Zhanhao Liu
University of Michigan
Ann Arbor, MI
zhanhaol@umich.edu

Qiulin Fan
University of Michigan
Ann Arbor, MI
rynnefan@umich.edu

1 Introduction

1.1 Background and Motivation

Early research on Natural Language to SQL (NL2SQL) before the LLM era relied on task-specific encoderdecoder architectures such as IR-Net(Guo et al., 2019), Seq2SQL(Zhong et al., 2017), SQLNet(Xu et al., 2017), SyntaxSQL-Net(Yu et al., 2018), and RAT-SQL(Wang et al., 2021). These models achieved remarkable progress on benchmark datasets but remain fundamentally limited, since their parameters are typically under the order of tens of millions, without any large-scale pretraining on natural language corpora. As a result:

- They lack general language modeling ability and exhibit poor few-shot generalization, upon domains and databases they have not seen, even simple ones.
- When confronted with complex queries (e.g. multi-table Join), these models often fail to transfer learned knowledge.

In our baseline evaluations, we observed that such models, if not fine-tuned on the specific dataset, perform poorly even on simple benchmarks such as WikiSQL.

Fortunately, with the emergence of large-scale LLMs, complex NL2SQL tasks has become more feasible. General-purpose models such as GPT-5-Codex demonstrate strong natural language to code reasoning. However, this shift also exposes practical limitations. Research this year(Liao et al., 2025) points out current SOTA methods largely depend on closed-source LLMs combined with prompt engineering, while open-source models still struggle on complex queries involving multiple joins or nested subqueries. In addition, large LLMs entail massive computational cost and memory consumption, making them impractical for domain-specific fine-tuning.

Therefore, our project aims to explore how a small-parameter LLM (3 ~8 B) can be fine-tuned to achieve strong NL2SQL performance comparable to large models in handling of complex tasks, but with a fraction of the computational cost.

While LearNAT(Liao et al., 2025) framework employs task decomposition, abstract syntax tree (AST) encoding and margin-aware reinforcement learning, our work instead focuses on graph-based schema modeling to strengthen the models structural reasoning. We hypothesize that incorporating graph representations provides a complementary advantagehelping smaller LLMs internalize multi-table relationships and bridge the gap between structural understanding and natural-language semantics.

1.2 Task Definition

The specific NLP task this project will address is multi-table Natural Language to SQL (NL2SQL) generation. The task of multi-table NL2SQL generation is to take a natural language query as input and automatically produce a syntactically correct and semantically accurate SQL query that retrieves the correct result from a relational database containing multiple interconnected tables.

- Input: a pair of $(\text{SQL schema}, \text{NL query})$. The schema may contain multiple tables with foreign key relationships. (Note: A foreign key is a column, or set of columns, in one table that refers to the primary key of another table.)
- Output: a structured SQL query that can be executed on the target database to return the intended result.

2 Data

2.1 WikiSQL Dataset

The **WikiSQL** dataset, released by Salesforce along with their paper *Seq2SQL: Generating Struc-*

tured Queries from Natural Language Using Reinforcement Learning, is one of the earliest and most widely used benchmarks for NL2SQL models. It is easy to extract and work with, containing 15,878 natural language (NL) to SQL pairs along with 4,550 data tables on which queries can be executed.

Using this dataset, we evaluated our generated SQL queries in two ways: 1.by directly comparing the predicted SQL with the gold (reference) SQL.2. by executing both queries on the actual table using DuckDB and comparing their outputs.

However, the WikiSQL dataset focuses solely on generating SQL queries over a single table. It does not include joins or nested queries, making it relatively simple. Since our project aims to improve models ability to generate SQL across multiple tables and databases—which is a key limitation of many current NL2SQL systems—we also incorporated the Spider dataset for a more realistic and challenging evaluation.

Examples from the WikiSQL Dataset

Example 1
Question: What institution had 6 wins and a current streak of 2?
GOLD SQL: `SELECT "Institution" FROM "table" WHERE "Wins" = 6 AND "Current Streak" = '2'`

Example 2
Question: Capital of Brze nad Bugiem has what population (1931) in 1,000s?
GOLD SQL: `SELECT "Population (1931) in 1,000s" FROM "table" WHERE "Capital" = 'Brze nad Bugiem'`

2.2 Spider Dataset

The **Spider** dataset, created by Yale University (*Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task*), covers a wide range of domains and includes many complex SQL queries involving multi-table joins and nested structures. Its development set contains 1,023 questions and gold SQL queries across 166 databases, each provided with full schema information.

Compared to WikiSQL, Spider introduces substantially higher difficulty, as queries often involve multiple tables, foreign key reasoning, and advanced SQL operators such as GROUP BY, ORDER BY, and nested subqueries. This makes Spider a more realistic benchmark for assessing compositional generalization and schema understanding.

To use this dataset, we preprocessed each database schema into textual form by concatenating table and column names along with their relationships, then reorganized the format for compatibility with our model. We evaluated our system against the `Text2SQL-1.5B` model and achieved an exact match accuracy of 41.3% on SQL generation. While the performance demonstrates reasonable generalization, further improvements may require enhanced schema linking and reasoning across complex relational structures.

```
{
    "db_id": "entrepreneur",
    "query": "SELECT T2.Date_of_Birth
        FROM entrepreneur AS T1
            JOIN people AS T2 ON T1.
                People_ID = T2.
                People_ID
            WHERE T1.Investor = 'Simon
                Woodroffe'
                OR T1.Investor = 'Peter
                    Jones'",
    "question": "Return the dates of
        birth for entrepreneurs who have
    either the investor Simon Woodroffe
        or Peter Jones."
}
```

Listing 1: Example from Spider dataset

For example of Spider schema structure, please see Appendix A.1.

3 Related Work

This section summarizes three lines of related work that are closely connected to our project: (1) traditional pre-LLM models for NL2SQL; (2) structurally enhanced models that incorporate schema and syntax information; and (3) recent LLM-based pipelines that achieve state-of-the-art performance through task decomposition and reinforcement learning.

Pre-LLM Models for NL2SQL. Early research on text-to-SQL adopted sequence-to-sequence architectures without large-scale language pretraining. Seq2SQL (Zhong et al., 2017) first introduced a reinforcement-learning objective to directly optimize execution accuracy rather than token-level similarity, but the approach suffered from unstable reward signals. SQLNet (Xu et al., 2017) improved upon this by using a sketch-based decoder to avoid

Table 1: Truncated table for Example 1 from Wikisql dataset (partial columns shown)

Institution	Wins	Losses	Home Wins	Home Losses
Boston College Eagles	6	1	3	1
Clemson Tigers	9	5	4	3
Duke Blue Devils	12	2	5	0
Florida State Seminoles	6	8	4	3
Georgia Tech Yellow Jackets	4	9	3	2
Maryland Terrapins	10	4	5	1
...	...	...	...	...

reinforcement learning altogether, achieving more stable training on the WikiSQL dataset. However, both methods focused on single-table scenarios and lacked the ability to generalize to complex, cross-domain databases.

Structure-Enhanced Text-to-SQL Models. Subsequent work emphasized the importance of modeling structural dependencies in database schemas and SQL syntax. SyntaxSQLNet (Yu et al., 2018) leveraged a syntax tree decoder to enforce SQL grammar constraints, enabling generation of compositional queries. IRNet (Guo et al., 2019) introduced intermediate representations to capture the semantic alignment between natural language and database elements, improving cross-domain transfer. RAT-SQL (Wang et al., 2021) further advanced this direction by proposing a relation-aware transformer encoder that explicitly encodes schema linking and foreign-key relations. Despite these advances, all such models remain relatively small in scale (tens of millions of parameters) and lack general-purpose language understanding, resulting in weak few-shot generalization and limited performance on multi-table join queries.

LLM-based NL2SQL and LearNAT. In the era of large language models, NL2SQL research has shifted toward leveraging general-purpose LLMs with prompt-based reasoning. LearNAT (Liao et al., 2025), a framework that substantially improves the NL2SQL performance of open-source LLMs through task decomposition and reinforcement learning. LearNAT decomposes complex SQL generation into structured subtasks using Abstract Syntax Trees (ASTs), combining three key components: (1) an AST-guided decomposition synthesis procedure that generates valid subtasks, (2) margin-aware reinforcement learning that optimizes multi-step reasoning with AST-based preference signals, and (3) adaptive demonstration retrieval during inference. Experiments on Spider and BIRD show that LearNAT enables a 7B-parameter open-source model to approach GPT-4-level accuracy, demonstrating the effectiveness of decomposition and RL-based supervision.

While LearNAT focuses on task decomposition through AST structures, our project extends this idea in a complementary direction by incorporating graph-based schema representations. Instead of decomposing SQL syntax, we explicitly model relational structures within the database schema as a graph to strengthen the models understanding of multi-table connections. This approach aims to enhance small-parameter LLMs structural reasoning ability in NL2SQL tasks without the computational overhead of large-scale reinforcement learning.

4 Methodology

4.1 Graph-Modeling Designs

Given a relational database schema that contains multiple tables, foreign keys, and column names, the central question is how to transform this schema into a graph structure that an LLM can effectively understand.

Basic structures include Table-level Graph: Nodes correspond to tables, and edges represent explicit foreign-key relationships.

```
Nodes: [Student, Course, Department]
Edges: Student -- Course (student_id)
       Course -- Department (dept_id)
```

Listing 2: Example of Table-Level Graph

and Column-level Graph: Nodes correspond to columns. Edges are constructed between foreign key columns and their referenced primary keys, and columns within the same table (intra-table edges).

```
[Student.id] -- [Course.student_id]
[Student.name] -- (intra) -- [Student.
    age]
```

Listing 3: Example of Column-Level Graph

Basic Design: Table&Column-level Hybrid Graph. We make hybrid of these two structures, let both tables and columns be represented as nodes. Edges include:

- table column containment edges
- foreign key connections between columns
- tabletable edges for cross-table relationships

```
Table: Student
   id, name, age
Table: Course
   cid, title, student_id
Edges: Student.id -- Course.student_id
```

Listing 4: Hybrid Graph

This design naturally expresses hierarchical structure and supports cross-table reasoning such as which tables contain columns related to Student. It strikes a balance between expressiveness and length, and is used as our main experimental design. Meanwhile, we plan to add more experimental features:

Extension 1: Semantic Edge. Built on top of the hybrid structure, this extension adds semantic edges between columns or tables whose names are semantically similar. We compute embedding similarity between names and connect pairs whose cosine similarity exceeds a threshold (e.g., 0.8):

```
[Birthday]  [DOB]
[Department]  [Dept]
```

Listing 5: Semantic Edge

This enriches schema understanding by introducing latent semantic connections that are not explicitly defined in the database schema, allowing the model to generalize across naming variations. However, the added edges may also increase graph density and introduce potential noise. We use this variant to evaluate the trade-off between structural richness and model robustness.

Extension 2: Typed Graph. As a further extension, all nodes and edges are annotated with type labels to explicitly distinguish their relational roles:

- Edge types: `foreign_key`, `intra_table`, `semantic_similar`.
- Node types: `table`, `column`, `primary_key`, `foreign_key`.

An example of the linearized input is shown below:

```
[table] Student
  [column_primary] id
  [column] name
  [column] age
[foreign_key_edge] Student.id -> Course.
    student_id
[semantic_edge] Birthday ~ DOB
```

Listing 6: Typed Graph

The typed graph provides the most expressive structural representation, enabling the LLM to differentiate between relationship types explicitly. However, this design also increases input length and may raise computational cost during fine-tuning. It will be evaluated as an advanced configuration in our ablation experiments.

In future experiments, we will operate on the fundamental graph design, together with two extensions, combining the results for comparisons.

4.2 Graph Encoding Methods

We will try two methods for integrating graph structures into LLM inputs and choose the better one in practice.

Text Linearization Graph structures are converted into textual prompts that describe table and column relations explicitly:

```
Schema Graph:
Table: Student(id, name, age)
Table: Course(cid, title)
Foreign Key: Student.id -> Course.
    student_id
Semantic Link: (DOB)  (Birthday)

Question: "List the names of students
    taking math."
```

Listing 7: Text Linearization

This approach maintains full compatibility with existing LLM tokenizers and training pipelines.

Graph Embedding. A lightweight graph encoder such as a GNN encoder encodes the schema graph into a dense embedding vector, which is injected into the LLM using parameter-efficient methods such as LoRA. For practical fine-tuning, we use tokenized tag-style formatting for schema representation:

```
[Table] Student [Columns] id(PK), name,
    age
[Table] Course [Columns] cid(PK), title,
     student_id(FK->Student.id)
[Relation] Student.id = Course.
    student_id
```

Listing 8: Graph Embedding

Currently we are using Text Linearization.

4.3 LoRA Fine-tuning

We will fine-tune a small-parameter open-source LLM using LoRA. LoRA introduces low-rank matrices to the attention and feed-forward layers, allowing the model to learn task-specific adaptations while keeping the original weights frozen.

Candidate models include Llama-8B-Instruct, Mistral-7B, Phi-3-Mini (3.8B), and Qwen-1.5-7B.

The training configuration is as follows as an example:

- Base model: Llama-8B
- Fine-tuning method: LoRA
- Rank r: 32, α: 32, dropout: 0.05
- Learning rate: 5e-4
- Sequence length: 2048 tokens
- Epochs: 5 on Spider
- Optimizer: AdamW

We have not run the training since we are still applying for free GPUs. If free GPUs are not powerful enough, we consider renting GPUs on Runpod. Moreover, the configuration is to be modified later in practical training.

4.4 Reinforcement Learning Discussion

Full RL training in NL2SQL faces several practical challenges: high execution cost (each SQL must be run on a database), sparse rewards, unstable gradients, and large computational overhead. As reported by prior work(Liao et al., 2025), RL often yields marginal improvements (1 to 3%) with significant complexity.

To address these issues, we would consider not using full RL, but adopt Execution-Guided Decoding (EGD) as an alternative. EGD applies execution feedback only at inference time:

1. Use beam search to generate top-k SQL candidates.
2. Execute each query on the database.
3. Select the highest-probability query that is executable and returns the correct result.

Mentioned by (Wang et al., 2018), This approach provides 3 to 5% improvement in execution accuracy without additional training cost, serving as a practical, execution-aware decoding strategy.

4.5 Final Problem Formulation

Graph Representation of Schema Let a relational schema be modeled as a typed multi-relational graph

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{R}),$$

where each node $v \in \mathcal{V}$ is either a table or a column, and each edge $e = (u, v, r) \in \mathcal{E}$ carries a relation type $r \in \mathcal{R}$. We consider a set of edge types

$$\begin{aligned} \mathcal{R} = \{&\text{table\_column}, \text{foreign\_key}, \\ &\text{intra\_table}, \text{semantic\_similar}\} \end{aligned}$$

Typed adjacency can be represented by a stack of binary matrices $A^{(r)} \in \{0,1\}^{|\mathcal{V}|\times|\mathcal{V}|}$, one per relation r.

For the semantic extension, we induce edges by embedding similarity. Let $e(v)$ denote the textual name of node v and $\mathbf{z}(v) = \mathrm{Enc}(e(v))$ be its embedding. A semantic edge is added between u and v if

$$\frac{\mathbf{z}(u)^\top \mathbf{z}(v)}{\|\mathbf{z}(u)\| \, \|\mathbf{z}(v)\|} \geq \tau,$$

with threshold $\tau \in (0, 1)$ is a hyperparameter we will set later. Suppose usiing text linearization for integration, the training input is the concatenation

$$x = \mathrm{Concat}\big(q,\ \mathrm{SchemaText},\ s\big),$$

where q is the natural language question and $\mathrm{SchemaText}$ is a plain-text schema description.

Generation Model Let p_θ be a small-parameter LLM with parameters θ. Given input x and optional graph feature $\mathbf{h}_{\mathcal{G}}$, the model generates a SQL token sequence $y = (y_1, \ldots, y_T)$ with

$$p_\theta(y \mid x, \mathbf{h}_{\mathcal{G}}) = \prod_{t=1}^{T} p_\theta\big(y_t \mid y_{<t}, x, \mathbf{h}_{\mathcal{G}}\big).$$

Supervised Fine-tuning Objective Given a dataset $\mathcal{D} = \{(q_i, \mathcal{G}_i, y_i^\star)\}_{i=1}^{N}$, we minimize the token-level negative log-likelihood

$$\mathcal{L}_{\text{SFT}}(\theta) = -\sum_{i=1}^{N} \sum_{t=1}^{T_i} \log p_\theta\big(y_{i,t}^\star \mid y_{i,<t}^\star,\ x_i,\ \mathbf{h}_{\mathcal{G}_i}\big).$$

LoRA Parameterization We adopt LoRA for parameter-efficient tuning. For a weight matrix $W \in \mathbb{R}^{m \times n}$ in attention or MLP blocks, we learn a low-rank update

$$W' = W + \Delta W, \quad \Delta W = BA,$$

where $A \in \mathbb{R}^{r \times n}$, $B \in \mathbb{R}^{m \times r}$, and $r \ll \min(m, n)$. Only A, B are trainable while W is frozen.

Execution-aware Inference via EGD This part has not been set up, but we will consider it later. Define an execution oracle $\mathcal{E}(y)$ that returns a tuple (c, r), where $c \in \{0, 1\}$ indicates compilability and $r \in \{0, 1\}$ indicates execution correctness against the gold answer. At inference time we generate a candidate set

$$\mathcal{C}_k = \mathrm{BeamSearch}\big(p_\theta(\cdot \mid x, \mathbf{h}_\mathcal{G}), k\big),$$

evaluate $\{\mathcal{E}(y) : y \in \mathcal{C}_k\}$, and select

$$\hat{y} \in \arg\max_{y \in \mathcal{C}_k} \Big(r(y),\, c(y),\, \log p_\theta(y \mid x, \mathbf{h}_\mathcal{G})\Big),$$

lexicographically by r then c then model score. This execution-guided decoding improves execution accuracy without modifying training.

Evaluation Metrics We report exact match accuracy

$$\mathrm{EM} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}\big[y_i = y_i^\star\big],$$

and execution accuracy

$$\mathrm{EX} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}\big[\mathcal{E}(y_i) = (1, 1)\big].$$

For multi-table reasoning we additionally measure join correctness

$$\mathrm{JAcc} = \frac{1}{N} \sum_{i=1}^{N} \frac{\big|\mathrm{Joins}(y_i) \cap \mathrm{Joins}(y_i^\star)\big|}{\big|\mathrm{Joins}(y_i) \cup \mathrm{Joins}(y_i^\star)\big|}.$$

5 Evaluation and Results

Table 2: Evaluation results of baseline NL2SQL models on different datasets.

Dataset	Model	Accuracy (%)
Spider	`gaussalgo/T5-LM-Large-text2sql-spider`	41.3
WikiSQL	`mrm8488/t5-base-finetuned-wikiSQL`	56.0

To evaluate our NL2SQL system, we primarily use the **Spider** benchmark, as it exhibits the properties we aim to improve upon. As described in the data section, we preprocess the **Spider** dataset to generate the correct table schemas for our experiments. For evaluation, our approach directly compares the predicted SQL code with the gold (reference) SQL, while ignoring differences in spaces and minor variations such as the use of double quotes (“ ”) versus single quotes (’ ’), since the functional equivalence of SQL queries is more important for practical use cases.

Due to the complex nature of the NL2SQL task, there are no simple random baselines. Therefore, for the **Spider** dataset, we selected `gaussalgo/T5-LM-Large-text2sql-spider`, a medium-sized model that is not state-of-the-art but achieves relatively strong performance. The `gaussalgo/T5-LM-Large-text2sql-spider` model is a fine-tuned version of Googles **T5-Large** (Text-to-Text Transfer Transformer) language model, specifically adapted for the **Spider** text-to-SQL benchmark. Built on the encoder–decoder architecture of T5, it converts natural language questions into structured SQL queries by framing the task as a sequence-to-sequence generation problem. We evaluated the `gaussalgo/T5-LM-Large-text2sql-spider` model on the **Spider** dataset and achieved a score of 41.3% accuracy.

Nevertheless, we also use the **WikiSQL** dataset to ensure that our model does not experience a performance drop on simpler NL2SQL tasks while focusing on multi-table generation. For evaluation, we adopt the same metrics as used for the **Spider** dataset, comparing the predicted SQL queries with the gold SQL and ignoring minor differences such as spacing or quotation marks. We evaluated the baseline model `mrm8488/t5-base-finetuned-wikiSQL` on this dataset and achieved an accuracy of 56%. The `mrm8488/t5-base-finetuned-wikiSQL` model is a **T5-base** transformer fine-tuned on the **WikiSQL** benchmark, designed to translate natural language questions into SQL queries over single-table databases.

6 Work Plan

Our project spans a total of seven weeks, of which the first two have already been completed. In the initial phase, we conducted background research, finalized the methodology, selected datasets (Spider and BIRD), and evaluated several baseline NL2SQL models to establish reference performance. The remaining five weeks will focus on implementation, fine-tuning, and evaluation, following the plan below:

- **Weeks 1 & 2 (Completed):** Conducted literature review and model planning.

  Defined the overall methodology and model architecture, including graph-based schema design and reinforcement learning discussion.

  Collected datasets (Spider and BIRD) and tested baseline models for initial performance comparison.

- **Week 3 (In-progress):** Implement the schema-to-graph conversion module using the Hybrid Graph baseline.

  Generate serialized graph representations for Spider databases and verify preprocessing correctness.

- **Week 4:** Integrate graph-augmented inputs with Llama-8B.

  Begin LoRA fine-tuning on the Spider dataset, adjusting learning rate and rank parameters for stability.

- **Week 5:** Extend experiments to include emantic Edge and Typed Graph variants.

  Compare their structural expressiveness and measure their influence on join reasoning accuracy.

- **Week 6:** Try to implement and test Execution-Guided Decoding (EGD) for inference, if everything went smoothly.

  Evaluate models on Spider dev and BIRD subsets using metrics such as Exact Match (EM), Execution Accuracy (EX), and Join Accuracy (JAcc).

- **Week 7:** Conduct full analysis and ablation studies across graph designs.

  Integrating all conclusions into final report and presentation slides.

We prioritizes completing the Hybrid Graph baseline and LoRA fine-tuning as the main deliverables.

References

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

Weibin Liao, Xin Gao, Tianyu Jia, Rihong Qiu, Yifan Zhu, Yang Lin, Xu Chu, Junfeng Zhao, and Yasha Wang. 2025. Learnat: Learning nl2sql with ast-guided task decomposition for large language models. *Preprint*, arXiv:2504.02327.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2021. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. *Preprint*, arXiv:1911.04942.

Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov, and Rishabh Singh. 2018. Robust text-to-sql generation with execution-guided decoding. *Preprint*, arXiv:1807.03100.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sql-net: Generating structured queries from natural language without reinforcement learning. *Preprint*, arXiv:1711.04436.

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

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

A Supplemental Material

A.1 Examples of Data

```
{
    "db_id": "entrepreneur",
    "foreign_keys": [
        [2, 6]
    ],
    "primary_keys": [1, 6],
    "table_names": [
        "entrepreneur",
        "people"
    ],
    "table_names_original": [
        "entrepreneur",
        "people"
    ]
},
{
    "column_names": [
        [-1, "*"],
        [0, "conductor id"],
        [0, "name"],
        [0, "age"],
        [0, "nationality"],
        [0, "year of work"],
        [1, "orchestra id"],
        [1, "orchestra"],
        [1, "conductor id"],
        [1, "record company"],
        [1, "year of founded"],
        [1, "major record format"],
        [2, "performance id"],
        [2, "orchestra id"],
        [2, "type"],
        [2, "date"],
        [2, "official ratings (millions)
            "],
        [2, "weekly rank"],
        [2, "share"],
        [3, "show id"],
        [3, "performance id"],
        [3, "if first show"],
        [3, "result"],
        [3, "attendance"]
    ],
    "column_names_original": [
        [-1, "*"],
        [0, "Conductor_ID"],
        [0, "Name"],
        [0, "Age"],
        [0, "Nationality"],
        [0, "Year_of_Work"],
        [1, "Orchestra_ID"],
        [1, "Orchestra"],
        [1, "Conductor_ID"],
        [1, "Record_Company"],
        [1, "Year_of_Founded"],
        [1, "Major_Record_Format"],
        [2, "Performance_ID"],
        [2, "Orchestra_ID"],
        [2, "Type"],
        [2, "Date"],
        [2, "Official_ratings_(millions)
            "],
        [2, "Weekly_rank"],
        [2, "Share"],
        [3, "Show_ID"],
        [3, "Performance_ID"],
        [3, "If_first_show"],
        [3, "Result"],
        [3, "Attendance"]
    ]
}
```

Listing 9: Example of Spider schema structure

A.2 Code for baseline

```
!rm -rf spider
!git clone https://github.com/taoyds/
    spider.git

import json

# Input: Spider-style schema list
IN_PATH = "/content/spider/
    evaluation_examples/examples/tables.
    json"
OUT_PATH = "tables_by_db.json"

def main():
    # Load the list of database schema
        dictionaries
    with open(IN_PATH, "r") as f:
        schema_list = json.load(f)

    # Convert list  dict keyed by db_id
    db_dict = {entry["db_id"]: entry for
         entry in schema_list if "db_id"
         in entry}

    # Save the dict as JSON
    with open(OUT_PATH, "w") as f:
        json.dump(db_dict, f, indent=2)

    print(f"Converted {len(db_dict)}
        databases into {OUT_PATH}")

if __name__ == "__main__":
    main()

# --- installs (for Colab / first time)
    ---
# !pip install -q transformers datasets
    tqdm

import re
import json
from typing import Optional, List

import torch
from datasets import load_dataset
from transformers import AutoTokenizer,
    AutoModelForSeq2SeqLM
from tqdm import tqdm

# ========================
# 1. Config & device
# ========================

MODEL_NAME = "gaussalgo/T5-LM-Large-
    text2sql-spider"
DEVICE = "cuda" if torch.cuda.
    is_available() else "cpu"

TABLES_JSON_PATH = "tables_by_db.json"
MAX_EXAMPLES: Optional[int] = 1034
```

```python
print("Using device:", DEVICE)

# ==========================
# 2. Load Spider dataset
# ==========================

ds = load_dataset("xlangai/spider")
train = ds["train"]
dev = ds["validation"]

if MAX_EXAMPLES is not None:
    dev = dev.select(range(MAX_EXAMPLES)
        )

print(f"Loaded {len(dev)} dev examples")

# ==========================
# 3. Load schema dict
# ==========================

with open(TABLES_JSON_PATH, "r") as f:
    db_schemas = json.load(f)
print(f"Loaded schema dict for {len(
    db_schemas)} databases")

def build_schema_string(db_id: str) ->
    str:
    """Convert a single db_id schema
        dict into a readable schema
        string."""
    db = db_schemas.get(db_id)
    if db is None:
        return ""
    table_names = db.get("
        table_names_original", [])
    col_names = db.get("
        column_names_original", [])
    col_types = db.get("column_types",
        [])
    primary_keys = set(db.get("
        primary_keys", []))

    from collections import defaultdict
    cols_by_table = defaultdict(list)
    for col_idx, (t_idx, col_name) in
        enumerate(col_names):
        if t_idx == -1:
            continue
        dtype = col_types[col_idx] if
            col_idx < len(col_types)
            else "text"
        cols_by_table[t_idx].append((
            col_idx, col_name, dtype))

    table_strings = []
    for t_idx, t_name in enumerate(
        table_names):
        parts = [f"\"{t_name}\""]
        for col_idx, col_name, dtype in
            cols_by_table[t_idx]:
            parts.append(f" \"{col_name
                }\" {dtype} ,")
        pk_names = [
            col_names[pk_idx][1]
            for pk_idx in primary_keys
            if col_names[pk_idx][0] ==
                t_idx
        ]
        if pk_names:
            parts.append("primary key: "
                + ", ".join(f"\"{n}\""
                for n in pk_names))
        table_strings.append(" ".join(
            parts))
    return " [SEP] ".join(table_strings)

# ==========================
# 4. Load model & tokenizer
# ==========================

tokenizer = AutoTokenizer.
    from_pretrained(MODEL_NAME)
model = AutoModelForSeq2SeqLM.
    from_pretrained(MODEL_NAME)
model.to(DEVICE)
model.eval()

@torch.no_grad()
def generate_sql(question: str, db_id:
    str, max_new_tokens: int = 128) ->
    str:
    """Generate SQL given a question and
        db_id, using its schema."""
    schema_str = build_schema_string(
        db_id)
    if schema_str:
        input_text = f"Question: {
            question} Schema: {
            schema_str}"
    else:
        input_text = f"Question: {
            question} Database: {db_id}"
    inputs = tokenizer(
        input_text,
        return_tensors="pt",
        truncation=True,
        max_length=512,
    ).to(DEVICE)
    outputs = model.generate(
        **inputs,
        max_new_tokens=max_new_tokens,
        num_beams=4,
        early_stopping=True,
    )
    return tokenizer.decode(outputs[0],
        skip_special_tokens=True).strip
        ()

# ==========================
# 5. Normalize & evaluate
# ==========================

def normalize_sql(sql: str) -> str:
    sql = sql.strip().rstrip(";")
    sql = re.sub(r"\s+", " ", sql)
    sql = sql.lower()
    sql = sql.replace('"', "'")
    sql = sql.replace(" ", "")
    return sql

lf_correct = 0
n = 0
inspect: List[dict] = []

for ex in tqdm(dev):
    question = ex["question"]
    db_id = ex["db_id"]
    gold_sql = ex["query"]
```

```
    pred_sql = generate_sql(question,
        db_id)
    lf_ok = normalize_sql(pred_sql) ==
        normalize_sql(gold_sql)

    if lf_ok:
        lf_correct += 1
    if len(inspect) < 10:
        inspect.append({
            "question": question,
            "db_id": db_id,
            "gold_sql": gold_sql,
            "pred_sql": pred_sql,
            "lf_ok": lf_ok,
        })
    n += 1

acc = lf_correct / n if n else 0
print(f"\nEvaluated on {n} dev examples"
    )
print(f"Logical-form accuracy: {acc:.4f}
    ")

print("\n=== Sample predictions ===")
for i, item in enumerate(inspect, 1):
    print(f"\nExample {i}")
    print("DB:", item["db_id"])
    print("Question:", item["question"])
    print("GOLD SQL:", item["gold_sql"])
    print("PRED SQL:", item["pred_sql"])
    print("LF match:", item["lf_ok"])
```

Listing 10: Code for baseline model evaluation on Spider