Investigating Entity Resolution through SQL Views in Text-to-SQL Models

Jess Moorefield
Department of Computer Science
Florida State University
Tallahassee, FL
moorefie@cs.fsu.edu

***Abstract*—This project explores using SQL views generated by object-relational mapping (ORM) as input to a Text-to-SQL model. This novel technique is intended to ease entity resolution, the key to a model's understanding of how database objects can be described in natural language. I constructed a software pipeline to automatically generate views to represent the entities in a given database and generate corresponding training data to be fed into an encoder-decoder model. My undergraduate thesis provides an in-depth explanation of leveraging object-relational mapping as the cornerstone of creating the views [1]. While experimental results were inconclusive, I discuss factors that may have influenced the model's performance and future directions for the technique.**

This project is submitted to the Department of Computer Science in partial fulfillment of the requirements for the degree of Master of Computer Science.

***Index Terms*—Text-to-SQL, entity resolution, natural language interfaces for databases**

I. Introduction

Text-to-SQL is a longstanding area of interest due to its important implications for data accessibility. Interacting with a database requires knowledge of SQL, or another query language, and most users who rely on data from a database do not have the requisite technical background to directly query it for information. Moreover, writing SQL queries also requires a user to understand how information concepts are represented in the database (i.e., the schema.) If a user does not have experience with SQL or undergo on the job training to learn, interactions with the database must be mediated through another person or interface, such as a database administrator or a service ticket request form. This delays fulfillment of the user's information need and creates a barrier between them and the data. Creating a tool that provides direct read-access to the database is a powerful step in lowering this barrier for non-technical users, allowing them to manage the data they rely upon while freeing SQL experts from servicing individual user requests.

II. Problem Statement

Differences between the way a user understands information in their domain (e.g., medicine, education, law, etc.) and the way that information is stored in a database is a major problem when creating tools for non-technical users. That is, because a user may describe their information need in natural language differently than the database schema representation, it is challenging to map the two together. This is referred to as 'entity resolution' or 'schema linking.' Hand-in-hand with this is the challenge of determining if a user's request requires information that is dispersed across multiple tables, and therefore understanding which tables need to be joined and on which columns. These two areas have been the central focus of my project, and I worked to address these challenges through the functionality of a SQL view.

The rationale behind this approach is that a view is an abstraction of a table in a database, and as such can centralize the related information about a particular entity in the database. Functionally, a view may be constructed for each entity by aggregating all the joins between tables where relevant information is dispersed. The 'universe' of information in the database can thus be reorganized into representations that are more intuitive for the entity resolution process. By enumerating the entities in a database and representing them with a view rather than disparate tables, we can create a one-to-one mapping between a concept in natural language and a SQL object. This simplifies the process of retrieving information about an entity by removing the need to join multiple tables (each of which naturally represent different concepts) together.

Concretely, from a query syntax perspective, substituting a view for a series of tables also decreases the need for join statement(s), as the joins have effectively become embedded in the 'FROM' clause as the view name. My project's central hypothesis is that simplifying both the syntactical complexity of the SQL query and the entities named in it will make it easier for the model to generate both valid and correct SQL to meet the user's information need.

III. Motivating Example

As a concrete example of this problem, consider a database for a hospital. This database includes information about patients, physicians, medical interventions, hospital infrastructure, and administrative records. The database also stores relationships within this information, such as what medication a physician prescribed to a patient during an appointment and which nurses are on call on which floors. These relationships are formalized by associative tables with multiple foreign keys.

A user might ask the question, "Which patients' prescriptions are written by physician John Smith?" She is asking about the patient entity, but the `Patient` table does not

contain all of this information. It stores the patient's name, and the physician's name is stored in the `Physician` table. However, since one patient can see many physicians, and one physician treats many patients, the patient name column is not also stored on the `Physician` table, and vice versa. To get from one table to the other, the model must use the `Prescribes` table, which stores the names of a particular patient being prescribed a particular medication by a particular physician.

Although the user's information need is related to single entity—a patient—the desired information about the entity is dispersed across multiple tables. Thus, the SQL query must join the three tables together:

```
SELECT patientName FROM Patient AS T1
JOIN Prescribes AS T2
ON T1.patientName = T2.patientName
JOIN Physician AS T3
ON T2.physicianName = T3.physicianName
WHERE T3.physicianName = 'John Smith'
```

To produce this query, the model must understand that 1) a join is required and 2) how to join the correct tables together to represent the entity the user is asking about.

IV. Methods

My project's approach is best described as a data prepossessing step to a Text-to-SQL model. The input data for the model is straightforward: a database schema and a corpus of English questions with corresponding SQL statements. In my project, these are represented by a SQL script and JSON documents, respectively.

A. Dataset

The dataset I used is called Spider [2], which is considered the standard benchmark for the Text-to-SQL task. Spider is a cross-domain dataset and includes 200 different databases across topics such as sports, employment, education, medicine, and more. In addition to the database schemas, the dataset also includes approximately 10,000 pairs of SQL queries and English questions. I selected a subset of Spider databases to work on while still preserving its cross-domain nature. My approach's specific modifications were to the database schemas and SQL training queries, where I replaced the original tables with corresponding views. I refer to this as the 'Views' variant of Spider.

The training data is split into three JSON files: train_spider.json, dev.json, and tables.json. The former two store the training and development datasets, respectively, and the latter encodes schema information about all the tables in the databases.

In the original Spider training data files, the JSON objects that represent each data point contain many properties, allowing for flexibility in designing a model's data input process. For simplicity, my project used a subset of these properties. train_spider.json and dev.json store a list of 3-tuples, each containing the database name, an English question, and the corresponding SQL query. tables.json stores a list of dictionary objects, with each database represented by a dictionary. It contains the names of every view and those names stripped of underscores; the column names of every view and those names stripped of underscores; and the types of every column.

Spider was created with the SQLite database management system (DBMS), and I also converted it for Microsoft SQL Server. However, I only evaluated the model on the SQLite data, as the SQL Server driver was incompatible with the Linux operating system on the machine I used to run the model.

Generating data for two different commercial DBMSs gave me insight into how this data processing might be adapted in a real-world enterprise setting. Organizations each have their own preferred tools for data management, and my project was designed with flexibility in mind to build a system that integrates with a variety of environments.

In the following section, I describe the scripts I wrote and sequenced into a software pipeline to take in the original dataset and transform it into the Views variant. All scripts were written in the Python programming language, and development was conducted on a Windows 10 virtual machine.

B. Scripts

1) Generate SQL Views: The first step of the pipeline is to use an ORM engine to generate the SQL views. This was the work of my undergraduate thesis. As a brief summary, I used the Microsoft Entity Framework ORM to generate object models for each table in a database. These models are C# classes that encode structural components of the database schema, such as foreign-key relationships, as C# properties. I then developed an algorithm that effectively reverse engineers a database entity by precomputing joins and join conditions for all the tables holding information about that entity. The result is the same number of SQL views as tables, but with each view aggregating all column fields reachable from that table through its foreign keys. That is, for every property in an object file, the script returns the column fields from that property's object file and then adds them into the current view.

I note that the algorithm only reaches a maximum depth level of 1, as it does not also search the foreign keys of the property's object file. Searching to depth level 1 retrieved all the columns named in a training query for every query in the dataset except one. This came from the `Dorm_1` database, whose schema includes a `Student` table, a `Dorm` table, a `Dorm_Amenity` table, and two associative tables, `Has_Amenity` and `Lives_In`. `Has_Amenity` represents the relationship of which dorms have which amenities, and `Lives_In` represents the relationship of which students live in which dorms.

The training query below answers the request, "Find the name of amenities of the dorm where the student with last name Smith is living in.'

```
SELECT T3.amenity_name FROM Dorm AS T1
JOIN Has_Amenity AS T2
ON T1.dormid = T2.dormid
JOIN Dorm_Amenity AS T3
ON T2.amenid = T3.amenid
JOIN Lives_In AS T4
ON T4.dormid = T1.dormid
JOIN Student AS T5 ON T5.stuid = T4.stuid
WHERE T5.lname = 'Smith'
```

The join between `Has_Amenity` and `Lives_In` must be accomplished through joining `Dorm_Amenity` in the middle. Thus, there is no direct foreign key relationship between `Lives_In` and `Dorm_Amenity`. In order for the algorithm to aggregate the columns from these two tables into a single view, it would need to search to depth 2. Thus, the resulting SQL view converted query still required one join:

```
SELECT amenity_name FROM nli.Lives_In AS
T1 INNER JOIN nli.Has_Amenity AS T2
ON T1.dormid = T2.dormid
WHERE lname = 'Smith'
```

All views were created as part of a schema named `nli` in each database, an acronym for 'natural language interface.' Clearly, the algorithm still significantly reduced the syntactical complexity of the query. In effect, this shifts the complexity of query generation from runtime to design time by compressing complex join conditions into a view rather than generating them on the fly.

In order to ensure that this corner case could still be answered using a SQL view, views were generated for all associative tables, even though they might not translate as intuitively as an entity as tables that represent an item or person.

2) Convert Training Queries: The next step was to modify the SQL training queries to use the views instead of the original tables. I stripped the queries from the JSON files and then wrote an algorithm to clean them. It removed any aliases and the join syntax. To select the appropriate FROM table, it iterated through all tables in the join sequence to select the first one that contains all of the columns mentioned in the query.

Note that this algorithm was intended as a first-pass, automated process. Some manual changes to the view queries were made to correct any that did not initially produce the same output as the original queries. I also mention that during this process, it was sometimes necessary to change the type of the join between two tables in a view. The view generation script already contained criteria to choose between an INNER and LEFT join, but in practice, some of the queries would not return the correct number of rows with the automatically chosen type.

In the case where multiple views contained all the columns named in a query, the corresponding view for the FROM table was selected. If it did not contain all the columns, then the first table mentioned in the original JOIN sequence with all those columns was selected. Importantly, the original English questions for the queries were preserved.

3) Populate New Training Data Files: Using the newly converted queries, I matched them to their English questions and stored them in a new train_spider.json file, and did the same for the dev.json data. The new tables.json file replaced the schema information about tables with information about the views.

In initial experimental runs, I kept the lone query with a JOIN statement in the new training dataset. However, this caused substantial confusion in the model, as it began generating JOIN clauses in other queries and predicting incorrect and incomplete join conditions. All experiments discussed in this paper did not include this query.

C. Model Architecture

I chose to modify RESDSQL [3], the highest-ranking encoder-decoder on the Spider leaderboard as of the time of writing. However, since my approach addresses the data preprocessing step rather than the model architecture itself, it can also be applied to large language models (LLMs) in future work. With the introduction of commercial LLMs such as GPT-4, the Text-to-SQL area has seen a recent surge of interest. GPT-4-based models have risen to the top of the official Spider leaderboard [4], with prompt engineering as a central technique, such as chain-of-thought reasoning.

I reworked RESDSQL's data processing pipeline to accept the Views variant and tweaked its evaluation process accordingly. One advantage of RESDSQL is that some of its scripts provide a boolean option to add or not add foreign key information about the database schemas. I set these options to false and removed all additional accesses to primary or foreign key information. As none of the queries inputted into the model contained join statements, I hypothesized that information about the keys would no longer be needed.

V. Experiments

The Spider dataset has two categories of evaluation: exact match, meaning whether two queries are syntactically equivalent, and execution accuracy, meaning whether two queries return the same data. RESDSQL is evaluated on both, and I compared the results from training the model on the Views variant with the results of training RESDSQL on the same databases from the original, "clean" Spider dataset. Importantly, I compared my results with the original RESDSQL code, preserving the foreign and primary key accesses.

A. Compute Limitations

RESDSQL's authors evaluated their model on a server with one NVIDIA A100 (80G) GPU and 256 GB memory [3]. I ran my experiments on a machine also with an A100 GPU and 125 GB memory. In addition, I shared the machine with other students, which varied the resources available to me at any given time. These memory limitations were a primary

challenge to my project. I had to significantly decrease the batch size hyper-parameter from its original value, both to run the model 'out of the box' and to run my experiments. As such, my experiments could not be compared to RESDSQL's best performance.

B. Original Model vs. Views Model

The first experiment simply compared the model's performance on a subset of the original training data versus its performance on the Views variant. The batch size was set to 9. The original model produced 51.2% exact match accuracy and 61.1% execution accuracy. The views model produced 21.0% exact match accuracy and 51.3% execution accuracy.

This significant decrease in both exact match and execution accuracy performance was unexpected. I analyzed the errors in the predicted queries and found that the overwhelmingly most common category was generating incorrect column names. A major contributor to this problem is that many column names overlap in the views. In some database schemas, there is also a mixture of singular and plural uses of the same noun, such as 'pet' and 'pets', which can also cause ambiguity. To better understand what factors might be influencing performance, I first approached the problem by using stemming.

C. Original Model vs. Stemmed Views Model

For this experiment, I stemmed the names of all columns and all the views. The batch size was again 9. The stemmed views model produced 8.2% exact match accuracy and 22.3% execution accuracy. The most common error category was also incorrect column name(s). Given that performance sharply degraded after stemming, it seemed that the problem might be due to the duplicate names of columns in the views.

Since the views were automatically generated by the ORM, the only modifications made up until this point were to the type of the joins between tables, as previously mentioned.

D. Original Model vs. Handcrafted Views Model

As a new experiment, I modified the original views by eliminating any columns with duplicate names but preserved differently-named columns that held the same information. The rationale behind these 'handcrafted' views was to give the model as much information about how that same piece of data could be described in natural language while minimizing the amount of unique columns the model needs to learn.

Note that this experiment did not include stemming, and the batch size was decreased to 7. To ensure a fair comparison, I ran the original model again on the new batch size. The original model produced 46.2% exact match accuracy and 55.3% execution accuracy. The handcrafted views model produced 25.2% exact match accuracy and 46.2% execution accuracy.

On the following page is a table summarizing the results statistics across the three experiments. Notably, in comparison with the original model with the same batch size, the handcrafted views had both the lowest exact match and execution accuracy differences, at -9.1% decrease and -21% decrease, respectively. This supports the hypothesis that the model would perform best on the minimum set of unique columns. Across all three experiments, the most common error in the predicted queries was 'no such column.'

VI. Discussion

These results suggest that schema linking relies upon the primary and foreign key information. However, it is not immediately clear what the root cause is for performance degradation. It might be due to a problem with the specific model chosen rather than the technique itself. RESDSQL might not be the optimal model for training on my approach's data, and another, differently designed model might be a better match. Given the amount of time necessary to study a model's code to understand how the training data is used and make appropriate modifications to the code, it was not feasible to select another model with a different design and compare its performance under the time constraints of my project.

Another factor may be that using views decreases query complexity but increases ambiguity. As such, a model might require additional training data or additional learning cues in the training data for better performance using cross-domain training. This is based on the idea that our entity views are deeply domain dependent and abstract commonalities between domains away.

Furthermore, one of the limitations I encountered was converting enough data to evaluate my approach. As other models use the full Spider dataset, a fair comparison with their top performance would require the same size dataset. Although my pipeline automatically generates and executes the views against the database and converts the original training queries to use views, there is still manual effort involved. Since this is not feasible for one person on a dataset the size of Spider, we cannot rely upon a model to learn by encountering large volumes of data.

VII. Future Directions

Cross-domain training hypothesizes that by looking at many different database schemas, the model will eventually be able to generalize to ones it has not seen before, because it has already trained on another database similar to it. As a future direction, within-domain training could offer new advantages. It better suits smaller datasets and allows us to investigate the effects of varying the size of the training set. Of particular interest is determining the lowest amount of training data for reasonable performance.

While cross-domain training would allow a model to be trained on an arbitrary database given a pre-existing corpus of training data, this is not necessarily a requirement for practical applications of Text-to-SQL. I propose that within-domain training suits scenarios in which corporate or government entities are dealing with niche situations as well as situations where an online model should be refined using questions and answers from the same system. Because every organization has a unique data management system, the most effective model would be the one that is optimized for the specific database platform, schema, and data within an organization.

TABLE I
EXPERIMENTAL RESULTS

Accuracy Metric	Original Model	Non-Stemmed Views	Stemmed Views	Handcrafted Views
Exact Match	51.2 / 46.2	21.0	8.2	25.2
Execution Accuracy	61.1 / 55.3	51.3	22.3	46.2
Batch Size	9 / 7	9	9	7

Moreover, in a real-world setting, mature organizations such as banks, hospitals, schools, and large corporations already accumulate institutional knowledge that can be utilized as training data for a model. SQL queries are continually being generated as employees put together reports and perform data analysis. Because an employee writing the report knows the natural language explanation for each query he/she executes, it is possible to build training data over time through employees' natural workflows. Many interesting research questions fall out of this reality as well, including: At what point does gradually accumulated institutional knowledge become sufficient training data for a model? Does continued refinement of a model naturally lead to overfitting? And what amount of change in the database structure or contents necessitates retraining?

To specifically address the question of minimal training data requirements, a future experiment could focus on the database in the Views variant with the largest amount of training data. This database's domain is the aforementioned hospital example and includes 15 tables and 164 pairs of natural language inquires and corresponding SQL queries. We could run a series of experiments varying the amount of data from the golden set given to the model for training compared to validation. We could then analyze both exact match and execution accuracy with decreasing amounts of training data to determine a minimal value to achieve a reasonable level of performance. My hypothesis is that the relationship between the size of the dataset and performance is proportional.

Given the smaller set of data within a single database and that my project's approach natively reduces the representation of entities, I expect the need for more varied training data to explain ambiguity that arises between overlapping column names and tangential database entities. This is in some ways orthogonal to what Spider provides, as the training queries in that dataset are specifically focused on cross-domain training.

VIII. CONCLUSIONS

This project has focused on evaluating a proof-of-concept model using a novel representation of ORM-generated views. From the beginning, its design and implementation has been motivated by how to extend itself into real-world use. Thus far, my work has included:

- Reorganizing the 'universe' of information in a database into distinct entities represented by SQL views,
- Building a software pipeline to automatically generate a set of views from a database schema file,
- Creating a variant of the Spider dataset to use these views instead of the original tables, and
- Evaluating the performance of the leading encoder-decoder model for Spider on our dataset.

This approach is more concerned with the data used to train a Text-to-SQL model rather than optimizing the model's performance. Because the Text-to-SQL task is motivated by real-world information needs, my project's ultimate contribution is in knowledge representation driven by practical application rather than evaluation in a lab setting.

REFERENCES

[1] J. Moorefield, "Leveraging Object-Relational Mapping to Reduce Entity Impedance in Natural Language Interfaces for Relational Databases," B.S. thesis, College of Arts and Sc., Florida State Univ., Tallahassee, 2022. [Online]. Available: https://diginole.lib.fsu.edu/islandora/object/fsu:802551/datastream/PDF/view

[2] T. Yu et al., "Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task," Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Jan. 2018, doi: https://doi.org/10.18653/v1/d18-1425.

[3] H. Li, J. Zhang, C. Li, and H. Chen, "RESDSQL: Decoupling Schema Linking and Skeleton Parsing for Text-to-SQL," Proceedings of the AAAI Conference on Artificial Intelligence, vol. 37, no. 11, pp. 13067–13075, Jun. 2023, doi: https://doi.org/10.1609/aaai.v37i11.26535.

[4] LILY (Language, Information, and Learning at Yale) Lab, Department of Computer Science, Yale University, "Spider: Yale Semantic Parsing and Text-to-SQL challenge", https://yale-lily.github.io/spider (accessed Dec. 11, 2023).