Accurate and Robust Text-to-SQL Parsing using Intermediate Representation

Yujian Gan

Doctor of Philosophy

Supervised by: Matthew Purver, John R. Woodward

School of Electronic Engineering and Computer Science
Queen Mary University of London

2022

Abstract

Text-to-SQL studies how to translate natural language descriptions into SQL queries. The key challenge is addressing the mismatch between natural language and SQL queries. To bridge this gap, we propose an SQL intermediate representation (IR) called Natural SQL (NatSQL), which makes inferring SQL easier for models and improves the performance of existing models. We also study the robustness of existing models in light of schema linking and compositional generalization.

Specifically, NatSQL preserves the core functionalities of SQL while it simplifies the queries as follows: (1) dispensing with operators and keywords such as *GROUP BY, HAVING, FROM, JOIN ON*, which are usually hard to find counterparts for in the text descriptions; (2) removing the need for nested subqueries and set operators; and (3) making schema linking easier by reducing the required number of schema items. On Spider, a challenging text-to-SQL benchmark that contains complex and nested SQL queries, NatSQL outperforms other IRs and significantly improves the performance of several previous SOTA models. Furthermore, for existing models that do not support executable SQL generation, NatSQL easily enables them to generate executable SQL queries.

This thesis also discusses the robustness of text-to-SQL models. Recently, there has been significant progress in studying neural networks to translate text descriptions into SQL queries. Despite achieving good performance on some public benchmarks, existing text-to-SQL models typically rely on lexical matching between words in natural language (NL) questions and tokens in table schemas, which may render models vulnerable to attacks that break the schema linking mechanism. In particular, this thesis introduces Spider-Syn, a human-curated dataset based on the Spider benchmark for text-to-SQL translation. NL questions in Spider-Syn were modified from Spider, by replacing their schema-related words with manually selected synonyms that reflect real-world question paraphrases. Experiments show that the accuracy dramatically drops with the elimination of such explicit correspondence between NL questions and table schemas, even if the synonyms are not adversarially selected to conduct worst-case adversarial attacks [1]. We present two categories of approaches to improve the model robustness. The first category of approaches utilizes additional synonym annotations for table schemas by modifying the model input, whereas the second category is based on adversarial training. Experiments illustrate that both categories of approaches significantly outperform their counterparts

[1]Following the prior work on adversarial learning, *worst-case adversarial attacks* refers to adversarial examples generated by attacking specific models.

without the defense and that the approaches in the first category are more effective.

Based on the above study results, we further discuss the **E**xact **M**atch based **S**chema **L**inking (**EMSL**). EMSL has become standard in text-to-SQL: many state-of-the-art models employ EMSL, with performance dropping significantly when the EMSL component is removed. However, we show that EMSL reduces robustness, rendering models vulnerable to synonym substitution and typos. Instead of relying on EMSL to make up for deficiencies in question-schema encoding, we show that using a pre-trained language model as an encoder can improve performance without using EMSL, creating a more robust model. We also study the design choice of the schema linking module, finding that a suitable design benefits performance and interpretability. Our experiments show that better understanding of the schema linking mechanism can improve model interpretability, robustness and performance.

This thesis finally discusses the text-to-SQL compositional generalization challenge: neural networks struggle with compositional generalization where training and test distributions differ. In this thesis, we propose a clause-level compositional example generation method. We first split the sentences in the Spider text-to-SQL dataset into sub-sentences, annotating each sub-sentence with its corresponding SQL clause, resulting in a new dataset Spider-SS. We then construct a further dataset, Spider-CG, by composing Spider-SS sub-sentences in different combinations, to test the ability of models to generalize compositionally. Experiments show that existing models suffer significant performance degradation when evaluated on Spider-CG, even though every sub-sentence is seen during training. To deal with this problem, we modify a number of state-of-the-art models to train on the segmented data of Spider-SS, and we show that this method improves the generalization performance.

Acknowledgements

My first appreciation goes to my first supervisor, Prof. Matthew Purver, for his fantastic supervision, guidance, and encouragement. I am extremely grateful that you took me on as your student and have always been very patient, enthusiastic, and friendly to me. Also, I greatly appreciate my second supervisor, Dr. John R. Woodward, for his great feedback, continuous encouragement, and guidance. Additionally, I am very grateful to my former supervisor Dr. John H. Drake, who taught me scientific research methods and helped me find the research direction. I would thank Prof. Edmund K. Burke, Dr. John H. Drake, and Dr. John R. Woodward for giving me this opportunity to study at Queen Mary. Finally, I would like to thank my two examiners Prof. Maria Liakata and Dr. Weiwei Sun for their insightful feedback, which greatly improve the quality of the thesis.

I would like to thank everyone in the **Computational Linguistics Lab** and **OR Group** at QMUL. In particular, I want to thank my independent assessor, Prof. Massimo Poesio, and Dr. Simon Rawles, who gave me a lot of advice and help. There is a large group of staff and Ph.D. students that I need to acknowledge for lunch conversations and discussions, social gatherings, and for being great friends: Dr. Ravi Shekhar, Dr. Juntao Yu, Dr. Sachchida Nand Chaurasia, Dr. Xinwei Wang, Sha Wang, Pavel Reich, Aiqi Jiang, Pakawat Nakwijit and many others. I also want to thank my collaborators, Dr. Xinyun Chen, Qiuping Huang, Jinxia Xie, and Pengsheng Huang, who have provided a lot of help and suggestion for my research.

Lastly, my family deserves endless gratitude. My parents supported me in pursuing my Ph.D. far away from my hometown. They gave me countless help, care, and encouragement. My sister helped me take the responsibility of taking care of our parents. Thank you all for the strength you gave me. I love you all!

Contents

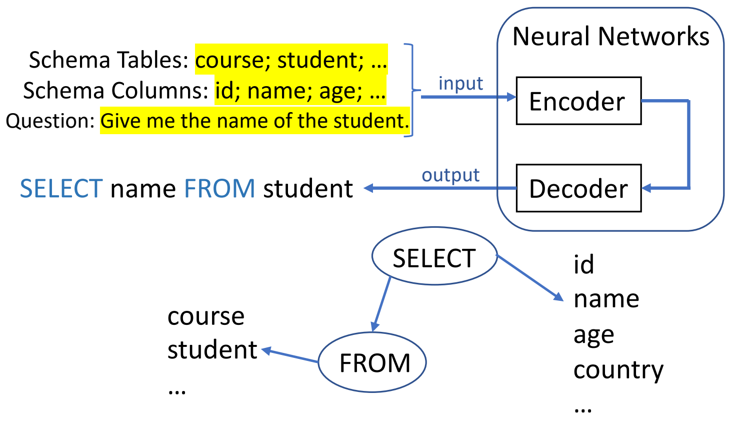
List of Figures

List of Tables

Chapter 1

Introduction

Text-to-SQL. With the development of the Internet, relational databases have become the mainstream to store large structural data from Internet. The relational database provides stable storage and convenient query functions for structural data. Although the relational databases can be efficiently accessed by skilled programmers via the structured query languages (SQL), a natural language interfaces to databases (NLDB) can facilitate the databases to be accessed by users without the knowledge of SQL. Therefore, text-to-SQL, which aims to translate the natural language (NL) descriptions/questions into SQL, has attracted attention from both industrial and academic communities. Currently, most text-to-SQL methods rely on neural networks, primarily based on the seq2seq [Sutskever et al., 2014] model structure, as for other methods we discuss in Chapter 3.1. Figure 1.1 shows a standard neural-based text-to-SQL pipeline.



Schema Tables: course; student; ...
Schema Columns: id; name; age; ...
Question: Give me the name of the student.
input
Neural Networks
Encoder
Decoder
output
SELECT name FROM student
SELECT
FROM
course
student
...
id
name
age
country
...

Figure 1.1: A standard text-to-SQL pipeline

In this pipeline, the model input is made up of the NL description/question and related schema tables and columns, while its expected output is the corresponding SQL. This pipeline simplifies some modules in the neural network model, such as word embeddings and attention. This pipeline illustrates that a successful text-to-SQL model requires the alignment of the NL with schema tables and columns and with the SQL keywords (SQL structure).

Text-to-SQL Assumption. Text-to-SQL assumes that the database schema is known, and the NL descriptions/questions inputted by the user must be a description of data query instructions. Although text-to-SQL is similar to the Question-Answering (QA) system in which the answer is the SQL query or the data queried by the SQL, the text-to-SQL does not assume to answer the NL questions that are not directly related to SQL instructions. Take Table 1.1 as a database table example. If you want to know whether Lily is older than 18, the existing text-to-SQL model does not allow directly inputting a question: 'Is Lily older than 18?'. SQL is used to access data in databases, not for QA, so you should input a question to query her age, such as: 'How old is Lily?'. Then check if she is over 18 by yourself. You can even ask the text-to-SQL model: 'What is the nationality of Lily who is older than 18?'. If no nationality data is returned, it means Lily is under 18, otherwise, she is over 18. To sum up, all allowable descriptions/questions are querying data from a database.

Student Information Table				
Name	Age	Gender	Nationality	Phone number
Lily	17	Female	UK	079453254
Lucy	18	Female	USA	085453254
Jack	18	Male	UK	079256354
Tina	19	Female	UK	079478412

Table 1.1: A database table containing four student information.

Cross-domain Text-to-SQL. The natural language interfaces to databases (NLDB) require semantic parsing models output corresponding SQL queries from the input NL question. If these models only generalize to new questions on the training domain, the NLDB cannot be adapted quickly to new databases, so it would not be widely used. Therefore, the model needs to generalize to new unseen databases with unseen questions. This is cross-domain text-to-SQL parsing.

SQL Intermediate Representation. Intermediate Representation (IR) is the data structure or any language used internally by a compiler or virtual machine to represent source language [Wikipedia contributors, 2022b]. In particular, an SQL IR is the language or data structure used internally to represent the SQL. SQL IR must be accurately converted to SQL and be independent of databases. SQL IR is usually designed for specific purposes, which are not easy to achieve with original SQL. For example, SQL IR in text-to-SQL is designed to bridge the gap between NL and SQL, making text-to-SQL easier to implement. Figure 1.2 shows a standard text-to-SQL pipeline based on IR, where the FROM clause in the IR has been removed, and the length of IR is shorter than that of SQL.

To synthesize SQL queries with more complex structures, IR was widely employed by the previous state-of-the-art (SOTA) models on the Spider dataset [Wang et al., 2020, Guo et al., 2019, Yu et al., 2018a, Shi et al., 2021]. Although the introduction of IR makes the whole pipeline one step longer than that in Figure 1.1, IR does reduce the difficulty of the model to predict the SQL because the complexity of IR is always less than or equal to the SQL. The IR to SQL conversion is determined by rules and does not require training.

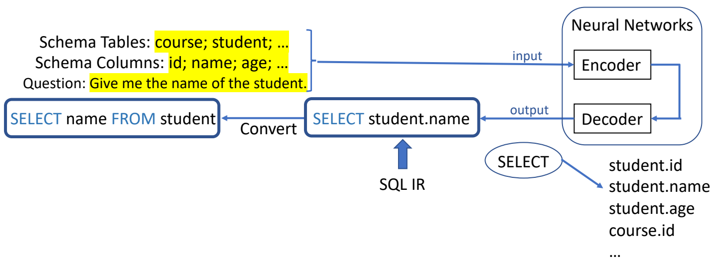
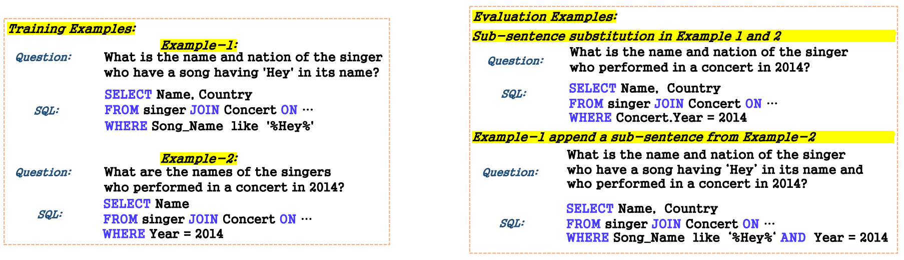


Figure 1.2: A standard text-to-SQL pipeline with IR

Robust Text-to-SQL Parsing. For a text-to-SQL model to be considered robust, either the testing error has to be consistent with the training error or the performance has to be stable (1) after adding some noise (2) after modifying the text or schema items to its synonym or (3) against compositional generalization.

Adding noise means inserting random characters in the NL question without changing its meaning. For example, we expected the model to predict the same SQL in Figure 1.1 when the noise '[ ]' was inserted into the question, such as 'Give me the name of the [student]'. Similarly, we expected the model could still predict the same SQL in Figure 1.1 when the question was modified to 'Give me the name of the highschooler'. If the model had failed to generate the correct SQL after question modification, i.e., the model failed to recognize the ‘highschooler’ as a ‘student’, we consider the model to be not robust against synonym substitution.

A robust text-to-SQL model also must have the ability to generalize compositionally. Compositional generalization is the ability to generalize to novel combinations of the components observed during training. Figure 1.3 presents the compositional generalization examples of text-to-SQL, where the left side provides the components in training. The components are sub-sentences and there are four components in the training set: (a) What is the name and nation of the singer; (b) who have a song having ‘Hey’ in its name?; (c) What are the names of the singers; (d) who performed in a concert in 2014?. The right side of Figure 1.3 presents the compositional examples, where the first example is the combination of (a) and (c), and the second example is the combination of (a), (b), and (c). We expected the model trained on the left side example to correctly predict the new sub-sentence combination examples on the right side. See Chapter 7 for how to get the components and generate the compositional examples.


Training Examples:

Example-1:

Question: What is the name and nation of the singer who have a song having 'Hey' in its name?

SQL:
```
SELECT Name, Country
FROM singer JOIN Concert ON ···
WHERE Song_Name  like  '%Hey%'
```

Example-2:

Question: What are the names of the singers who performed in a concert in 2014?

SQL:
```
SELECT Name
FROM singer JOIN Concert ON ···
WHERE Year = 2014
```

Evaluation Examples:

Sub-sentence substitution in Example 1 and 2

Question: What is the name and nation of the singer who performed in a concert in 2014?

SQL:
```
SELECT Name,  Country
FROM singer JOIN Concert ON ···
WHERE Concert.Year = 2014
```

Example-1 append a sub-sentence from Example-2

Question: What is the name and nation of the singer who have a song having ‘Hey’ in its name and who performed in a concert in 2014?

SQL:
```
SELECT Name,  Country
FROM singer JOIN Concert ON ···
WHERE Song_Name  like  ‘%Hey%‘ AND  Year = 2014
```

Figure 1.3: A compositional generalization example in text-to-SQL

Thesis Statement. This thesis studies machine learning models to perform text-to-SQL parsing, with a focus on SQL IR and model robustness. We claim that our work has improved model performance and robustness. Furthermore, we propose two datasets for evaluating model robustness against synonym substitution and compositional generalization, providing for a future study in text-to-SQL.

1.1 Thesis Contributions

This thesis has contributed to several aspects of text-to-SQL: SQL intermediate representation *(Natural SQL)*, evaluation methods, and improvement of robustness.

1.1.1 Natural SQL (NatSQL)

We present *Natural SQL (NatSQL)*, a new SQL intermediate representation (IR) that offers simplified queries over other IRs, while preserving a high coverage of SQL structures. More importantly, NatSQL further eliminates the mismatch between NL and SQL, and can easily support executable SQL generation. Figure 1.4 presents a sample comparison between NatSQL and other IRs. We observe that there is a mismatch between the NL word `‘and’` and the *INTERSECT* SQL keyword. To translate the NL question into a corresponding query, previous IRs need the models to distinguish whether the word `‘and’` corresponds to *INTERSECT*, not required for NatSQL. Among all IRs, NatSQL provides the simplest and shortest translation, while NatSQL structure also aligns best with the NL question.

NatSQL preserves the core functionalities of SQL, while simplifying the queries as follows: (1) dispensing with operators and keywords such as *GROUP BY, HAVING, FROM, JOIN ON*, which are usually hard to find counterparts for in the text descriptions; (2) removing the need for nested subqueries and set operators, using only one *SELECT* clause in NatSQL; and (3) making schema linking easier by reducing the required number of schema items normally not mentioned in the NL question. The design of NatSQL easily enables executable SQL generation, which is not naturally supported by other IRs.

We compare NatSQL with SQL and other IRs by incorporating them into existing open-source neural network models that achieve competitive performance on Spider. Our experiments show that NatSQL boosts the performance of these existing models, and outperforms both SQL and other IRs. In particular, equipping RATSQL+GAP [Shi et al., 2021] with NatSQL achieves a new state-of-the-art execution accuracy on the Spider benchmark. These results suggest that to improve the ability of text-to-SQL models to understand and reason about the NL descriptions, designing IRs to better reveal the correspondence between NL and query languages is a promising direction.

1.1.2 Evaluating Performance against Synonym Substitution and Compositional Generalization

The state-of-the-art models have achieved impressive performance on text-to-SQL tasks (e.g., around 70% accuracy on the Spider test set, even if the model is tested on databases unseen in training). However, we suspect that such cross-domain generalization heavily relies on exact lexical matching between the NL question and the table schema. As shown in Figure 1.5, names of tables and columns in the SQL query are explicitly stated in the NL question. Such questions constitute the majority of cross-domain text-to-SQL benchmarks, in-

Question :
Which film has more than 5 actors and less than 3 in the inventory?

SQL :
```
SELECT T1.title FROM film AS T1 JOIN film_actor AS T2 ON T1.film_id = T2.film_id GROUP
BY T1.film_id HAVING count(*) > 5 INTERSECT SELECT T1.title FROM film AS T1 JOIN
inventory AS T2 ON T1.film_id = T2.film_id GROUP BY T1.film_id HAVING count(*) < 3
```

The IR of RAT−SQL : (Remove the JOIN ON Clause)
```
SELECT title FROM film, film_actor GROUP BY film_id HAVING count(*) > 5
INTERSECT SELECT title FROM film, inventory GROUP BY film_id HAVING count(*) < 3
```

The IR of SyntaxSQL : (Remove the JOIN ON and FROM Clause)
```
SELECT film.title GROUP BY film.film_id HAVING count(*) > 5 INTERSECT
SELECT film.title GROUP BY film.film_id HAVING count(*) < 3
```

SemQL : (Remove the JOIN ON, FROM and GROUP BY Clause. Merge the HAVING and WHERE clause)
```
SELECT film.title WHERE count(film_actor.*) > 5  INTERSECT
SELECT film.title WHERE count(inventory.*) < 3
```

NatSQL : (Further remove the set operators based on SemQL)
```
SELECT film.title WHERE count(film_actor.*) > 5 and count(inventory.*) < 3
```

Figure 1.4: A sample question in Spider dataset with corresponding SQL and IRs.

cluding both Spider and WikiSQL. Although assuming exact lexical matching is a good starting point for solving the text-to-SQL problem, this assumption usually does not hold in real-world scenarios. Specifically, it requires that users have precise knowledge of the table schemas to be included in the SQL query, which could be tedious for synthesizing complex SQL queries.

We investigate whether state-of-the-art text-to-SQL models preserve good prediction performance without the assumption of exact lexical matching, where NL questions use synonyms to refer to tables or columns in SQL queries. We call such NL questions *synonym substitution* questions. Although some existing approaches can automatically generate synonymous substitution examples, these examples may deviate from real-world scenarios, meaning they may not follow common human writing styles or may even accidentally become inconsistent with the annotated SQL query. To provide a reliable benchmark for evaluating model performance with synonym substitution questions, we introduce Spider-Syn, a human-curated dataset constructed by modifying NL questions in the Spider dataset. Specifically, we manually replace the schema annotations in the NL question with synonyms while the corresponding SQL queries keep unchanged, as shown in Figure 1.5. We demonstrate that when models are only trained on the original Spider dataset, they suffer a significant performance drop on Spider-Syn, even though the Spider-Syn benchmark is not designed to exploit the worst-case attacks for text-to-SQL models. It is therefore clear that the performance of these models will suffer with real-world use, particularly in cross-domain scenarios.

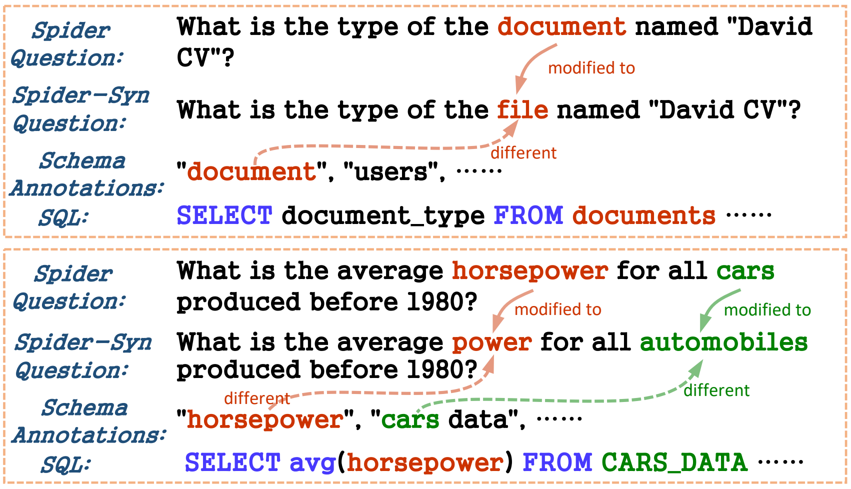

Spider Question:
What is the type of the document named "David CV"?
modified to
Spider-Syn Question:
What is the type of the file named "David CV"?
different
Schema Annotations:
"document", "users", ……
SQL:
SELECT document_type FROM documents ……
Spider Question:
What is the average horsepower for all cars produced before 1980?
modified to
modified to
Spider-Syn Question:
What is the average power for all automobiles produced before 1980?
different
different
Schema Annotations:
"horsepower", "cars data", ……
SQL:
SELECT avg(horsepower) FROM CARS_DATA ……

Figure 1.5: Sample Spider questions that include the same tokens as the table schema annotations, and such questions constitute the majority of the Spider benchmark. In our Spider-Syn benchmark, we replace some schema words in the NL question with their synonyms, without changing the SQL query to synthesize.

Except for the Spider-Syn, we also propose the Spider-CG dataset to evaluate model robustness against compositional generalization. We construct the Spider-CG (CG stands for *compositional generalization*) by substituting sub-sentences with those from other samples or composing two sub-sentences to form a more complicated sample. For example, the right-side examples in Figure 1.3 belong to the Spider-CG and are automatically generated from the left-side examples. We demonstrate that when models are trained only on the original Spider dataset, they suffer a significant performance drop on the Spider-CG, even though the domain in the training set.

1.1.3 Improvement of Robustness

To improve the robustness of text-to-SQL models against synonym substitution, we use synonyms of table schema words, either manually annotated or automatically generated when no annotation is available. We investigate two categories of approaches to incorporate these synonyms. The first category of approaches modify the schema annotations of the model input so that they align better with the NL question. No additional training is required for these approaches. The second category of approaches are based on adversarial training, in which we augment the training set with NL questions modified through synonym substitution. Both categories of approaches significantly improve the robustness, and the first category is effective and requires less computational resources.

To improve the model generalization ability, we introduce a sub-sentence-based text-to-SQL training paradigm. This paradigm requires models to encode the whole sentence but decode the sub-sentences one by one. Then the models collect all output SQL clauses to generate the final target SQL query. This paradigm improves the model generalization ability when evaluated in the Spider-CG and also improves the performance in the Spider benchmark. The model based on this paradigm is trained on the Spider-SS, a manually annotated sub-sentence to the SQL clause dataset.

1.2 Thesis Associated Publications

Portions of the work detailed in this thesis have been presented in national and international conferences, as follows:

- Yujian Gan, Matthew Purver, and John R. Woodward. A review of cross-domain text-to-SQL models. **In Proceedings of the 1st Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 10th International Joint Conference on Natural Language Processing: Student Research Workshop, December 2020**

- Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R. Woodward, John Drake, and Qiaofu Zhang. Natural sql: Making sql easier to infer from natural language specifications, **In Findings of the Association for Computational Linguistics: EMNLP 2021**

- Yujian Gan, Xinyun Chen, Qiuping Huang, Matthew Purver, John R. Woodward, Jinxia Xie, and Pengsheng Huang. Towards robustness of text-to-SQL models against synonym substitution, **In Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, 2021**

- Yujian Gan, Xinyun Chen, and Matthew Purver. Exploring underexplored limitations of cross-domain text-to-sql generalization. **In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2021**

- Yujian Gan, Xinyun Chen, Qiuping Huang, and Matthew Purver. Measuring and improving compositional generalization in text-to-sql via component alignment, **In Findings of the Association for Computational Linguistics: NAACL 2022**

- Yujian Gan, Xinyun Chen, and Matthew Purver. Re-appraising the Schema Linking Mechanism in Text-to-SQL, **Under Reviewed**

- Yujian Gan and Matthew Purver. Clause-based Modeling for Compositionally Generalizable Text-to-SQL Parsing, **Under Reviewed**

1.3 Thesis Structure

The thesis is structured as follows:

- **Chapter 2.** We introduce the background of text-to-SQL from four aspects: task description, machine learning methods, key modules for text-to-SQL, and baseline models.

- **Chapter 3.** We review previous work on semantic parsing and text-to-SQL, including text-to-SQL datasets, paradigms, and robustness. This chapter is based on [Gan et al., 2020].

- **Chapter 4.** We introduce NatSQL. We first compare the difference between NatSQL and other IRs. Experiments show that NatSQL improves the performance of several models and enables them to generate the executable SQL. This chapter is based on [Gan et al., 2021c].

- **Chapter 5.** We evaluate the robustness of text-to-SQL models with regard to synonym substitution. In particular, we introduce Spider-Syn, a human-curated dataset based on the Spider benchmark for text-to-SQL translation. Additionally, we present two categories of approaches to improve the model robustness. The first category of approaches uses additional synonym annotations for table schemas by modifying the model input, while the second category is based on adversarial training. We show that our first approach outperforms adversarial training methods on Spider-Syn, and achieves competitive performance on worst-case adversarial attacks. This chapter is based on [Gan et al., 2021a].

- **Chapter 6.** We extend the work of Chapter 5 to evaluate the exact match based schema linking (EMSL). Inspired by the low robustness of the model caused by EMSL in Chapter 5, we discuss whether EMSL can be removed and why previous researchers did not remove it. We found that EMSL introduces the vulnerability to the text-to-SQL models and can be replaced by better input encoding. This chapter is based on the sixth paper listed in Chapter 1.2.

- **Chapter 7.** We investigate the robustness of text-to-SQL models to compositional generalization. Inspired by the close relationship between NatSQL and NL, we first split the sentences in the Spider text-to-SQL dataset into sub-sentences, annotating each sub-sentence with its corresponding NatSQL clause, resulting in a new dataset: Spider-SS. We then construct a further dataset: Spider-CG, by composing Spider-SS sub-sentences in different combinations, to test the ability of models to generalize compositionally. Experiments show that existing models suffer significant performance degradation when evaluated with Spider-CG, even though every sub-sentence has been seen during training. To deal with this problem, we modify a number of state-of-the-art models to train on the segmented data of Spider-SS, and we show that this method improves the generalization performance. This chapter is based on [Gan et al., 2022].

- **Chapter 8.** We summarize and discuss future work. We highlight the findings and contributions with respect to the NatSQL and improvement of model robustness.

Chapter 2

Background

Automatic generation of SQL queries from natural language (NL) has been studied in the literature for a number of years Warren and Pereira [1982], Androutsopoulos et al. [1995], Ana-Maria Popescu et al. [2003], Li et al. [2006], Li and Jagadish [2014], Dong and Lapata [2018], Iacob et al. [2020]. Early works Warren and Pereira [1982], Androutsopoulos et al. [1995] focus on template-based or rule-based method, which cost heavy human engineering toward the goal. Besides, it is difficult to design various templates or rules in advance for different domains. Because of these shortcomings, current researchers tend to use machine learning methods. With the availability of large-scale training data and advances in deep learning, machine learning methods have made great progress in text-to-SQL parsing. A typical machine learning method for text-to-SQL parsing is the sequence to sequence (seq2seq) [Sutskever et al., 2014] model, which automatically learns a mapping function from the input NL question to the output SQL. The seq2seq based approaches leverage an encoder to encode the input NL questions together with related table schema into vectors and then use a grammar-based neural decoder to decode these vectors to generate the target SQL. Seq2seq provides an end-to-end way for training and has become the mainstream for text-to-SQL parsing.

In this chapter, we first introduce the text-to-SQL. Then, we provide the basic Machine Learning methods. Finally, we introduce some text-to-SQL models used in this thesis.

2.1 Text-to-SQL

2.1.1 Overview

SQL (Structured Query Language) is a language designed mainly for managing data stored in a relational database, including querying, adding, updating, and deleting data [Wikipedia contributors, 2022c]. SQL is widely used in handling structured data, i.e., data incorporating relations among entities and variables. SQL is designed considering the convenience of data management, where it can access or modify many records with one single command. Therefore, SQL is quite different from NL and cannot be easily used by non-professionals.

The SQL for querying data contain several key components: SELECT, FROM, WHERE, JOIN ON, GROUP BY, HAVING, subqueries, and set operators. We introduce them one by one below:

- **SELECT**: The SELECT clause is used to select data from a database and is an indispensable SQL clause for querying data.
- **FROM**: The FROM clause is used to specify which database to select, which is also indispensable. The FROM clause sometimes appears with the JOIN ON clause when it involves multiple tables.
- **WHERE**: The WHERE clause contains query conditions, which are used to filter records. The query condition format is column operator value, such as age >= 18 representing to filter out the records whose age column value is less than 18. There are fifteen condition operators: '*between*', '=', '>', '<', '>=', '<=', '! =', '*in*', '*like*', '*is*', '*exists*', '*not in*', '*not like*', '*not between*', and '*is not*'. The condition value can be a static value, column, or subquery. A WHERE clause contains one or more conditions connected by the logical conjunction ('*and*' and '*or*').
- **JOIN ON**: JOIN ON clause is used to combine rows from two or more tables, as mentioned in the FROM clause. The combination method can be conditional, and this condition is written after the ON keyword. The ON condition format is similar to the WHERE condition.
- **GROUP BY**: The GROUP BY clause is used in collaboration with the SELECT clause to group rows having identical values into summary rows. The GROUP BY clause is often used with aggregate functions. There are five aggregate functions: '*count()*', '*max()*', '*min()*', '*sum()*', and '*avg()*', which calculates a set of values, and returns a single value.
- **HAVING**: The HAVING clause is similar to the WHERE clause. Since the WHERE clause cannot be used with aggregate functions, the HAVING

clause is added to the SQL. The HAVING condition takes effect after the GROUP BY, while the WHERE clause is conducted before the GROUP BY.

- **Subquery**: The subquery is a full SQL SELECT query embedded in the main SQL SELECT query. The data returned from a subquery can be a condition value which means we can use a subquery to replace a static value in a condition.

- **Set Operators**: The set operators combine the results of two SQL SELECT queries into a single result. There are three set operators: UNION, INTERSECT, and EXCEPT, where the UNION returns all distinct rows selected by either query, the INTERSECT returns all distinct rows selected by both queries, the EXCEPT returns all distinct rows selected by the first query but not the second.

Text-to-SQL is essential for non-professionals to access the databases. Most text-to-SQL research focuses on data queries whose SQL starts with the SELECT keyword. Chapter 1 gives a brief text-to-SQL introduction and example, and we can find that there is a significant difference or mismatch between input (NL and table schema) and output (SQL). For better discussion, the next section provides a formal problem definition of text-to-SQL parsing.

2.1.2 Task Formulation

Symbol:	Description
$\mathcal{S}$	Sequence of database schema tokens, which consists of tables, columns and cell values.
$\mathcal{T}$	Sequence of table tokens.
$\mathcal{C}$	Sequence of column tokens.
$\mathcal{V}$	Sequence of cell value tokens.
$\mathcal{Q}$	Sequence of question tokens.
t	Table token.
c	Column token.
v	Cell value token.
q	Question token.
$\mathcal{I}$	Text-to-SQL input, which consists of question and schema.
$\mathcal{O}$	Text-to-SQL output, referring to SQL query.

Table 2.1: The notations used in this chapter.

Text-to-SQL [Qin et al., 2022] is to convert an NL question under the database schema to its corresponding SQL that can be executed in the same database. Table 2.1 provides formal notation to define task formulation. Existing text-to-SQL parsing research can be divided into two categories: single-turn and multi-turn settings. Formally, for the single-turn text-to-SQL parsing, given an NL question $\mathcal{Q}$ and the corresponding database schema $\mathcal{S} = <\mathcal{T}, \mathcal{C}, \mathcal{V}>$, the goal is to generate a SQL query $\mathcal{O}$. Specifically, the question $\mathcal{Q} = \{q_1, q_2, ..., q_n\}$

is a sequence of $\mathcal{Q}$ tokens. The database schema consists of table sequence $\mathcal{T} = \{t_1, t_2, ..., t_n\}$, table sequence $\mathcal{C} = \{c_1, c_2, ..., c_n\}$, and cell value sequence $\mathcal{V} = \{v_1, v_2, ..., v_n\}$. Each t_i stands for a table name that contains one or multiple words. Similarly, Each c_i stands for a column name that also contains one or multiple words. The cell value is the data stored in the database. Each v_i denotes a cell value that contains none or one or multiple words. Sometimes, the cell values $\mathcal{V}$ can be ignored or inaccessible. In this case, the database schema can be expressed as $\mathcal{S} = <\mathcal{T}, \mathcal{C}>$. The whole input of the task can be denoted as $\mathcal{I} = <\mathcal{Q}, \mathcal{S}>$.

For multi-turn text-to-SQL parsing, it is to repeat the single-turn task several times, where the NL questions may contain anaphora that refers to objects in the previous NL questions. Formally, let $\mathcal{U} = \{u_1, u_2, ..., u_n\}$ denote a sequence of single-turn text-to-SQL parsing task with n turns, where $\mathcal{U}_n = (\mathcal{I}_n, \mathcal{O}_n)$ represents the n-th parsing task which is the combination of a input $\mathcal{I}_n$ and a output SQL query $\mathcal{O}_n$. The input $\mathcal{I}_n$ can be denoted as $\mathcal{I}_n = <\mathcal{Q}_n, \mathcal{S}, \{u_1, ..., u_{n-1}\}>$. At the n-th turn, given an NL question $\mathcal{Q}_n$, the corresponding database schema $\mathcal{S}$, and the historical parsing tasks $\{u_1, ..., u_{n-1}\}$, the goal is to produce a SQL query $\mathcal{O}$.

2.1.3 Evaluation Metrics

The text-to-SQL tasks generally evaluate the methods by comparing the generated SQL queries against the ground-truth answers. The exact set match accuracy (EM) and execution (EX) [Yu et al., 2018b] accuracy are evaluation metrics for evaluating the single-turn text-to-SQL. For the multi-turn text-to-SQL, the evaluation metrics include question match accuracy (QM) and interaction match accuracy (IM) [Yu et al., 2019b].

Single-turn Text-to-SQL Evaluation

Exact Set Match Accuracy (EM). The exact set match accuracy is calculated by comparing the clauses (without values) in the predicted SQL query and the ground-truth SQL query. Before comparison, these two queries are parsed into normalized SQL clauses such as SELECT, GROUP BY, WHERE, ORDER BY, etc. The EM gives a positive result to the predicted SQL query only if all of the SQL clauses are correct by a set comparison as follows:

$$\text{score}(\hat{\mathcal{O}}, \mathcal{O}) = \begin{cases} 1, & \hat{\mathcal{O}} = \mathcal{O} \\ 0, & \hat{\mathcal{O}} \neq \mathcal{O} \end{cases}$$

where $\mathcal{O} = \{k_1, ..., k_n\}$ and $\hat{\mathcal{O}} = \{\hat{k_1}, ..., \hat{k_n}\}$ denote the SQL clause component sets of the ground-truth SQL query and the predicted query respectively. The k

here denotes a SQL clause, and n is the number of parsed components. Although condition values are inside some of the SQL clause k, the EM comparison ignores these values. Then, the exact set match accuracy is calculated by:

$$\mathrm{EM} = \frac{\sum_{i=1}^{N} \mathrm{score}\big(\hat{O}_i, O_i\big)}{N}$$

where N stands for the total number of samples. EM does not compare the values and can focus on whether the grammar of the predicted SQL is consistent with that of ground-truth. However, considering that an NL question may correspond to multiple SQL queries, the disadvantage of EM is that it cannot give positive results to the equivalent predictions.

Execution Accuracy (EX). We calculate the EX by comparing the query data of executing the ground-truth SQL and the predicted SQL query on the database. Unlike the EM does not need the condition values, to pass the EX, the predicted SQL must contain the correct condition values. EX gives positive the result to the predicted query as correct only if the data of executing the predicted and the ground-truth SQL query are the same:

$$\mathrm{score}(\hat{\mathcal{D}}, \mathcal{D}) = \begin{cases} 1, & \hat{\mathcal{D}} = \mathcal{D} \\ 0, & \hat{\mathcal{D}} \neq \mathcal{D} \end{cases}$$

where $\mathcal{D}$ and $\hat{\mathcal{D}}$ denote the data queried by the ground-truth SQL query and the predicted query, respectively. Similar to EM, we calculate the EX as follows:

$$\mathrm{EX} = \frac{\sum_{i=1}^{N} \mathrm{score}\big(\hat{D}_i, D_i\big)}{N}$$

EX also cannot give a positive result to partial equivalent SQL and may generate positives to incorrect predictions whose queried data from ground-truth SQL is empty.

Multi-turn Text-to-SQL Evaluation

In a multi-turn text-to-SQL setting, given a total of N samples, there are a total of $A = N * R$ questions where each sample contains R rounds.

Question Match Accuracy (QM). Before calculating the QM, we calculate the EM score over all questions. We calculate the EM score the same as the single-turn settings mentioned above. Then we calculate the question match accuracy as follows: The question match accuracy is calculated as the EM score over all questions. Its value is 1 for each question only if all predicted SQL clauses are correct. We first calculate the EM score for each question as follows:

$$\mathrm{QM} = \frac{\sum_{i=1}^{A} \mathrm{score}\big(\hat{\mathcal{O}}_i, \mathcal{O}_i\big)}{A}$$

where $A = N * R$ stands for the total number of questions.

Interaction Match Accuracy (IM). Similarly, before calculating the IM, we calculate the EM score first. Then we calculate the score of each interaction (sample) which is positive only if all the predictions within the interaction are correct. Formally, the score for each interaction (sample) is calculated by:

$$\text{interaction\_score} = \begin{cases} 1, & \prod_{i=1}^{R} \text{score}\left(\hat{\mathcal{O}}_i, \mathcal{O}_i\right) = 1 \\ 0, & \prod_{i=1}^{R} \text{score}\left(\hat{\mathcal{O}}_i, \mathcal{O}_i\right) = 0 \end{cases}$$

where R denotes the number of turns in each interaction (sample). Then, we calculate the IM score as follows:

$$\text{IM} = \frac{\Sigma_{i=1}^{N} \text{ interaction\_score }_i}{N}$$

where N denotes the total number of interactions (samples).

2.2 Machine Learning (ML)

Recent years have seen great progress on the text-to-SQL problem [Zhong et al., 2017, Dong and Lapata, 2018, Yu et al., 2018b, Guo et al., 2019, Bogin et al., 2019a, Wang et al., 2020], with neural networks having become the *de facto* approach. Therefore, we introduce ML approaches we need to know.

ML is the scientific study of algorithms and statistical models that computer systems use to effectively perform a specific task without using explicit instructions, relying on patterns and inference instead [Wikipedia-contributors, 2019]. Although there are many ML approaches, we only focus on some of them widely used by text-to-SQL, including recurrent neural networks (RNNs), the sequence to sequence (seq2seq) model, transformer, and graph neural networks.

2.2.1 Recurrent Neural Network

Researchers have been studying RNNs since the 1980s [Hopfield, 1982, Rumelhart et al., 1986]. An RNN is composed of cells with specific structures. There are many types of RNN cells and structures. Long short-term memory (LSTM), as one type of RNN cell, was invented by [Hochreiter and Schmidhuber, 1997] and it can get good performances in speech recognition [Sak et al., 2014], machine translation [Sutskever et al., 2014] and other language models [Jozefowicz et al., 2016]. The bidirectional recurrent neural network (BRNN) as a widely used RNN structure was introduced to increase the amount of input information available to the model [Schuster and Paliwal, 1997].

Long Short-Term Memory

Nowadays, the widely used LSTM has a with forget gate (also called 'keep gate') [Gers, 1999] and all references to LSTM in this thesis have such a gate.

The forget gate can be taught to weaken, even clear, the previous cell state (memory) in a certain state, so the LSTM can control the long-term memory through the forget gate.

Bidirectional Recurrent Neural Network

A Bidirectional recurrent neural network (BRNN) is the connection of two RNN in the opposite direction, as shown in Figure 2.1. There are only two RNN cells in Figure 2.1: the blue squares A and A'. The output of y is generated by repeated use of the squares A and A'.

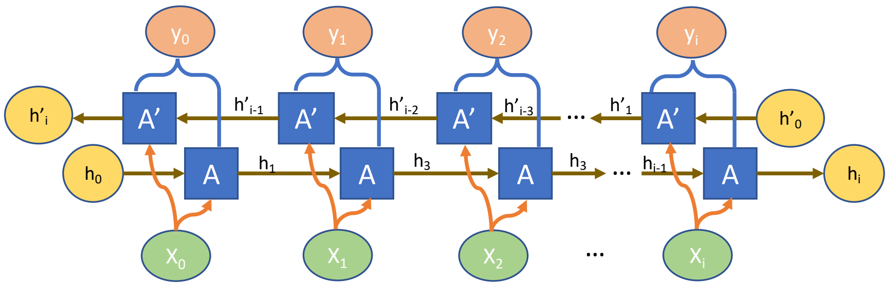


Figure 2.1: General Structure of Bidirectional Recurrent Neural Networks

Let us suppose the outputs of square A and A' are y_A and $y_{A'}$. So the final output y (shown in the orange circle in Figure 2.1) normally is the concatenation of y_A and $y_{A'}$, but you can also define the final y as equal to $y_A + y_{A'}$.

2.2.2 Sequence to Sequence Learning

Basic Sequence to Sequence Model

Seq2seq learning is an extremely powerful ML model that offers an end-to-end approach to sequence learning problems that makes minimal assumptions about sequence structure [Sutskever et al., 2014]. It was widely used in sequence learning problems such as machine translation [Luong et al., 2015], chat-bot [Qiu et al., 2017], text-to-SQL [Dong and Lapata, 2018], etc. Figure 2.2 gives an example of a chat-bot based on a seq2seq model such that the model input is a sentence from the user and the model output is the reply to the input. The seq2seq model consists of two parts: an encoder and a decoder. The blue squares of the encoder and green squares of the decoder in Figure 2.2 can be RNN cells.

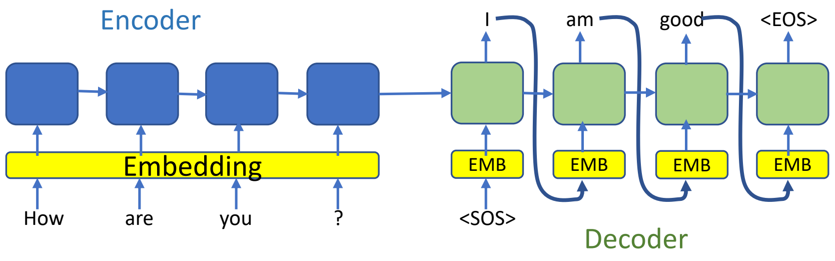


Figure 2.2: Sequence to Sequence Learning Model

Sequence to Sequence Model with Attention

The attention mechanism was invented by [Luong et al., 2015] and now is widely used in the seq2seq model. Attention not only improves the performance of the seq2seq model but also makes the learning results better explained. The attention mechanism can be considered a key-value process by which different keys can get different values. The key in here is the (input) hidden state of decoder cells.

Figure 2.3 gives an example of an attention-based model modified from Figure 2.2. Intuitively, in the basic seq2seq model without attention, given that the decoder only gets the h_0 generated by the encoder, which represents the whole sentence, it is more difficult for the decoder to learn how to output correct results than it is for the attention-based model. Because the attention mechanism tells the seq2seq model what part of the input it should pay attention to when generating each output, most current seq2seq models contain an attention structure by default.

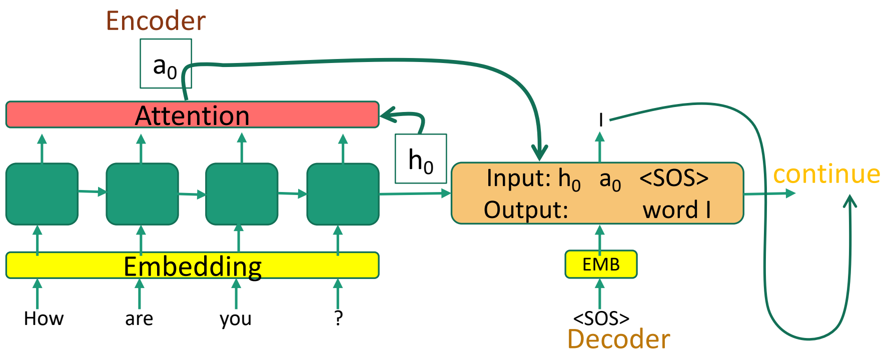


Figure 2.3: The attention-based model modified from Figure 2.2

Transformer

Vaswani [2017] proposed the transformer model that only keeps the attention mechanism dispensing with the RNN, where the transformer still belongs to the seq2seq encoder-decoder architecture. Unlike RNNs, the attention mechanism does not process data in order. However, given that the NL order is essential, the transformer generates position embeddings for every input token. Compared with RNNs, there are several primary benefits of transformer. The computations can be performed in parallel, running faster than RNNs. The improvement of computing speed makes it easier to build large-scale seq2seq models. Finally, the transformer has a better learning ability for longer sequences than RNNs because it is not sensitive to the sequence length.

Grammar Decoder

The standard seq2seq decoder output a sequence token. Although it can output a SQL sequence, the grammar of the output SQL may be incorrect. In order to be grammatically correct, researchers [Xiao et al., 2016, Cheng et al., 2017a, Yin and Neubig, 2017a] introduce the grammar decoder to ensure that the output follows the SQL grammar. The grammar decoder is widely used for complex text-to-SQL, replacing the original seq2seq decoder, given that the grammar decoder provides a constraint for the decoding process. Besides text-to-SQL, the grammar decoder is also a common module for other semantic parsing tasks.

Both the standard seq2seq decoder and grammar decoder generates output based on probabilities. Without the constraint, each step of the standard decoding may generate any SQL keywords. Although a standard decoder with high accuracy has a higher probability of generating SQL keywords that meet the grammatical rules, it is still possible to generate the SQL with grammar errors. The grammar decoder generates the contents following a predefined abstract syntax tree (AST) [Wang et al., 1997] that can be extracted from the SQL or other programming languages. AST act as a constraint, ensuring the next decoding step does not generate the grammar error SQL. For example, the standard decoder may output a *WHERE* clause following the *SELECT* clause without any constraint, as shown in Figure 2.4. This error does not occur with the grammar decoder because the SQL AST stipulates that the following clause of *SELECT* can only be *FROM*.

2.2.3 Graph Neural Networks

A graph neural network is a class of neural network for processing data best represented by graph data structures [Scarselli et al., 2009]. They were origi-

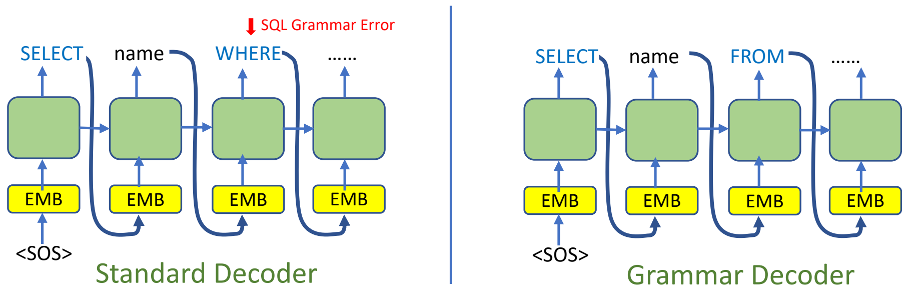


Figure 2.4: The standard decoder may generate grammar error SQL, but grammar decoder will not.

nally popularized by their use in supervised learning about properties of various molecules [Gilmer et al., 2017, Wikipedia contributors, 2022a]. In computer science, a graph is a data structure consisting of two components: nodes (vertices) and edges[Amal Menzli, 2021]. The schema database can be represented as a graph where the columns and tables are the nodes and the foreign keys are edges. We can get more edges by defining new relation, such as column-to-table edges, primary key edges and table-to-table edges. Normally, for text-to-SQL under single table, there is no need to use graph neural networks. If only one table, all edges for the columns are equivalent in a graph, meaning that there are only table-to-column edges. Graph neural networks cannot give different information or values for equivalent edges, which restricts their use in single table based text-to-SQL.

Currently, there are some text-to-SQL models using graph neural networks, such as GNN [Bogin et al., 2019a], Global-GNN [Bogin et al., 2019b], and LGESQL [Cao et al., 2021]. The GNN represents a schema as a graph and uses graph neural networks to embed each schema item. The GNN and Global-GNN are designed by the same author; the difference between them is that the Global-GNN use a global reasoning module to choose correct ambiguity schema items. In LGESQL, the graph nodes include both schema items and question tokens, while nodes in Global-GNN only contain the schema items. Therefore, the graph in LGESQL can encode the schema items together with the question words.

2.2.4 Adversarial Training

Adversarial training is the method of the attacks on machine learning models, and of the defenses against such attacks [Wikipedia contributors, 2022d]. Siva Kumar et al. [2020] exposes the fact that professionals report a dire need for protection methods of machine learning systems against different types of attacks. Most machine learning methods are designed for specific problems, under the assumption that the training and test data are sampled from the same distribution. However, this assumption often can be violated in practices, where users or hackers may purposely supply out-of-distribute data that violates the assumption. Adversarial training is introduced to avoid this problem [Szegedy et al., 2013]. There are mainly two steps in adversarial training. Firstly, adversarial training generates out-of-distribute samples to attack (test) the trained models. Then it collects the failure samples from the previous step for re-training the model. The two steps can be repeated several times. With more and more out-of-distribute samples added to the training set, the trained model becomes more robust.

2.3 Key Modules for Text-to-SQL

Recent ML-based text-to-SQL models mostly divide the problem into two subtasks: generating SQL keywords and filling the schema items, also named schema linking.

2.3.1 SQL Keywords Generation

ML-based text-to-SQL models rely on the decoder to generate the SQL Keywords. The decoder for text-to-SQL can be divided into two categories: sketch-based methods and generation-based methods [Qin et al., 2022].

The sketch-based methods predefine a class of classifiers to determine whether to generate a target SQL keyword. There are several models employing the sketch-based method, such as SQLNet [Xu et al., 2017], TypeSQL [Yu et al.], SQLova [Wonseok Hwang, Jinyeung Yim, Seunghyun Park, 2019] and Coarse2Fine [Dong and Lapata, 2018]. For example, SQLNet uses a classifier to determine that the AGG function is either an empty token or one of the aggregation operators, such as AVG and MIN. Generally, the construction of the sketch may be different for different models. For example, different from SQLNet, TypeSQL combines the select-column classifier and the where-column classifier into a single classifier since their input and output are similar. In the final stage, an execution-guided [Wang et al., 2018] decoding strategy often be utilized to prevent generating non-executable SQL queries. However, complex SQL generation tasks require sketch-based methods to define too many classifiers to work, which is complicated to implement. Thus, the sketch-based approaches are popular on the simple text-to-SQL task, such as the WikiSQL [Zhong et al., 2017] dataset, but are rarely used on complex text-to-SQL scenarios, such as the Spider [Yu et al., 2018b] dataset.

Recently, researchers employed generation-based approaches to handle the complex text-to-SQL task, building on the seq2seq model. Some works follow the standard seq2seq modeling, while others employ the grammar decoder. For example, Shaw et al. [2021a] and Scholak et al. [2021] leverage the pre-trained seq2seq model T5 [Raffel et al., 2019] finetuned on the text-to-SQL dataset for SQL generation. Although T5 neglects the SQL grammar during the decoding process, its large-scale pre-training data and big model size ensure the model's accuracy. Formally, the T5 decoder follows the standard text generation process, as shown in Figure 2.2. Since the standard seq2seq models may not generate SQL queries with correct grammar, some researchers choose the grammar decoder instead. In particular, the grammar decoder generates the contents following a predefined abstract syntax tree (AST) [Wang et al., 1997] that can be extracted from the SQL or other programming languages.

2.3.2 Schema Linking

Schema Linking Definition

To achieve good performance on text-to-SQL tasks, a neural model needs to correlate natural language queries with the given database schema, and we call this process as *schema linking*. Previous work often explicitly designs a module to perform the schema linking, and we name it as Exact Match based Schema Linking (*E*MSL) [Guo et al., 2019, Bogin et al., 2019a, Wang et al., 2020]. Specifically:

- **Schema linking** is the alignment between the entity references in the question and the schema columns or tables.
- A **schema linking module** is a trainable component that learns to perform schema linking, based on features that relate word tokens in the question to schema items.
- A **schema linking feature** encodes this relational information; e.g., it can represent the similarity between words in the question and schema items.
- **Exact match based schema linking (EMSL)** is a type of schema linking feature obtained by the exact lexical match between the words in the question and words in schema items.

Figure 2.5 illustrate the relations between these concepts, where Schema linking includes everything related to filling the schema items. We discuss the details of EMSL in Chapter 6.

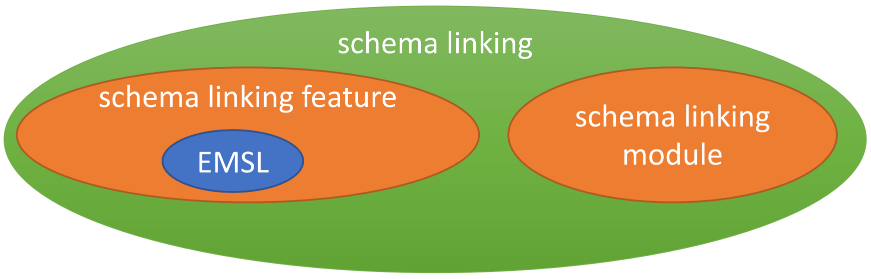


Figure 2.5: The schema linking family.

Schema Linking Construction

In general, schema linking establishes a link between question word tokens and schema items, where this value/weight guides the text-to-SQL model to choose the closest schema item. We refer to this value/weight the schema linking value. Any text-to-SQL model with decent performance needs a schema linking value. This value can be obtained in many ways, such as calculating the similarity between encoded question word tokens and schema items or directly obtaining the value through the EMSL feature.

Some works have implemented EMSL by recognizing the columns and the tables mentioned in a question before training the model [Guo et al., 2019, Bogin et al., 2019a, Wang et al., 2020]. It should be noted that Guo et al. [2019] and Wang et al. [2020] named the EMSL schema linking in their paper while Bogin et al. [2019a] did not mention this EMSL but implemented it in the code. EMSL was essential in these models because in the ablation study of IRNet [Guo et al., 2019] and RATSQL [Wang et al., 2020] based on Spider [Yu et al., 2018b], removing the EMSL caused the biggest performance decline when compared with removing other removable modules [Guo et al., 2019, Wang et al., 2020].

Additionally, the EMSL can be integrated with the database contents, improving the performance of IRNet, RATSQL and GNN models. For example, in Table 2.2, without inspecting the content of the database, it would be hard to construct a link between the word `houses` and the column '`property type code`', even by experts, given that the word `houses` might be a redundant word that often appears in questions.

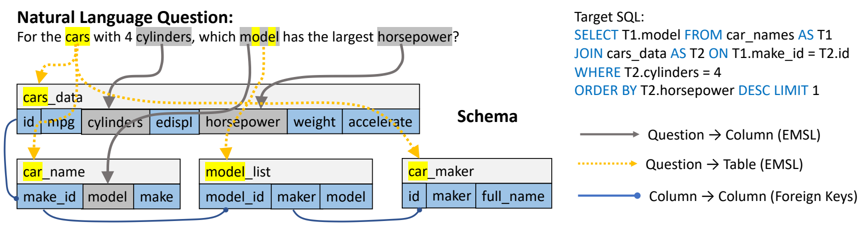
Question:	What are the names of houses properties?
SQL:	**SELECT** name **FROM** Properties **WHERE** type_code = 'House'

Table 2.2: An example of requirements for database content

Schema Linking Based on Graph

Figure 2.6 illustrates the challenge of ambiguity in schema linking while '*model*' in the question refers to `car_names.model` rather than `model_list.model`. Graph neural networks can give a bigger schema linking value to the `car_names.model` than `model_list.model` because the uniquely matched `horsepower` column can propagate its weight through the schema relations (e.g., foreign keys) to the `car_names.model`.

The other benefit of using a graph is giving the schema items that are not mentioned in the question a more significant schema linking value. Examples are often seen in *JOIN ON* clauses and subqueries. In Figure 2.6, it is hard to construct the schema linking from the question to the columns `cars_data.id` and `cars_names.make_id` that appear in the *JOIN ON* clause of the SQL. The graph neural networks can construct the schema linking value for these two columns from the propagation of other linked columns.



```
Target SQL:
SELECT T1.model FROM car_names AS T1
JOIN cars_data AS T2 ON T1.make_id = T2.id
WHERE T2.cylinders = 4
ORDER BY T2.horsepower DESC LIMIT 1
```

Figure 2.6: A challenging text-to-SQL example from the Spider dataset.

2.3.3 SQL Intermediate Representation (IR)

SQL IR is a language that bridges the NL and the SQL since the SQL is designed for accessing the database, not for human communication. Although SQL IR is not a must, it improves the model in both SQL keyword generation and schema linking and has been widely used in text-to-SQL models. Using IR, the text-to-SQL will become text-to-IR-to-SQL, where the original text-to-SQL model can be directly used in text-to-IR, as shown in Figure 1.2. For a more detailed introduction to IR, please refer to Chapter 1.

Generally speaking, SQL IR should be designed simpler than SQL, which means it will dispense with some SQL clauses. The deleted SQL clauses and their schema items must be generated when converting the SQL IR to SQL, which ensures that the SQL IR does not lose information. Different clauses require different generation methods. Take generating the JOIN ON clause as an example, their schema items tend to be the foreign keys that appear in the tables in the rest of the SQL clauses.

Early work on SQL IR tried to use an IR to translate an NL question and then convert it to SQL queries [Woods, 1978, Li and Jagadish, 2014]. Li et al. [2014] proposed an IR for SQL called Schema-free SQL, for users who did not need to know all of the schema information. The IR in SyntaxSQLNet [Yu et al., 2018a] represents an SQL statement without *FROM* and *JOIN ON* clauses. SemQL [Guo et al., 2019] removes the *FROM, JOIN ON* and *GROUP BY* clauses, and combines the *WHERE* and *HAVING* conditions. The IR in Edit-SQL [Zhang et al., 2019] also combines the *WHERE* and *HAVING* conditions but keeps the *GROUP BY* clause. IR is also used to improve compositional generalization in semantic parsing [Herzig et al., 2021a].

Yu et al. [2018a] introduced an SQL IR that dispense with *JOIN ON* and *FROM* clauses. This IR generate full SQL containing *JOIN ON* clauses in a deterministic way by analyzing the schema structure. However, this IR might not generate the correct *JOIN ON* clause when there were more than one available *JOIN ON* clause or they were facing the self-join.

Guo et al. [2019] further proposed an SQL IR, named SemQL that removes the *GROUP BY* clause and merges the *HAVING* and *WHERE* clauses. SemQL reduces the reasoning work from SQL structure generation that does not significantly benefit from graph neural networks. However, about 20% of the generated *GROUP BY* clauses from SemQL are different from the original, restricting its performance in exact set match metrics. Although most different *GROUP BY* clauses do not affect the accuracy of execution match metrics, SemQL cannot generate executable SQL in the current version. IR research still has room for improvement.

Compared to existing IRs for SQL, our NatSQL, introduced in Chapter 4, has further simplified the SQL language, moving closer towards bridging the gap between NL descriptions and SQL statements. SemQL is the closest to NatSQL in the above IRs, where NatSQL can be considered an IR that has further simplified the structure and improved the coverage by SemQL.

2.4 Text-to-SQL Models Used in the Thesis

We introduce three recent relatively high-performance text-to-SQL models used in this thesis. They are all open source and can easily be reproduced and modified. Table 2.3 compares their main design choices.

GNN [Bogin et al., 2019a]

The GNN model is built based on AllenNLP [Gardner et al., 2018] platform. To our knowledge, the GNN model is the first model for the Spider [Yu et al.,

Moldes	EMSL	Extra Schema Encoder	W_Emb	IR
GNN	✓	Graph Neural Networks	From Scratch	-
IRNet	✓	-	GLOVE / BERT	SemQL
RATSQL	✓	Relation-Aware Transformer	GLOVE / BERT	SQL without JOIN ON

Table 2.3: The baseline text-to-SQL models. The three models are based on the seq2seq structure. The Extra Schema Encoder indicates other encoders except for the seq2seq encoder. EMSL denotes the exact match based schema linking. W_Emb means word embeddings. From Scratch represents the model train its word embeddings from scratch. GLOVE [Pennington et al., 2014] and BERT [Devlin et al., 2019] are two widely used word embeddings.

2018b] complex text-to-SQL task using the graph neural networks to encode the whole schema database. As discussed in Chapter 2.2.3, the graph is suitable for representing the complex structure of schema database. The GNN model is the only one of the three models that learn to generate both SQL without SQL IR and to generate *JOIN ON* clauses. *JOIN ON* clause generation requires understanding the foreign key relationships between the schema items, while the graph neural networks can help. GNN uses the generation-based approach to generate the SQL query, employing a grammar decoder working on the full SQL AST. Although the authors did not mention the EMSL in their paper, the GNN model had used it. After that, the authors propose a re-ranking strategy [Bogin et al., 2019b] to select the best SQL query from the candidates predicted by the GNN model, improving the overall performance.

IRNet [Guo et al., 2019]

Similar to the GNN model, IRNet is also a generation-based model employing a grammar decoder and uses the EMSL to construct the schema linking. However, the IRNet generates SemQL instead of SQL. The SemQL is an SQL IR, briefly introduced in Chapter 2.3.3. The SemQL can be converted to the final SQL stably. Experiments show that SemQL can improve several previous text-to-SQL models. Besides the SemQL, the IRNet is equipped with the BERT [Devlin et al., 2019] to improve its performance. In schema linking, IRNet employs ConceptNet [Speer and Havasi, 2012b] to construct extra EMSL features. The standard EMSL can link words only by exact lexical match, so IRNet assembles the ConceptNet to extend the EMSL to a synonym match. Although there are no graph neural networks in IRNet, the overall performance of the IRNet is better than the GNN model.

RATSQL [Wang et al., 2020]

RATSQL is also a generation-based model using the grammar decoder and EMSL, and integrates the advantages of the previous two models. In RATSQL, you can find SQL IR, graph, and BERT. The SQL IR in RATSQL is pretty close to the SQL compared to SemQL, only dispensing with *JOIN ON* clause. Although it raises the learning difficulty, the SQL IR in RATSQL ensures higher coverage to the SQL than SemQL. There are no graph neural networks inside the RATSQL, but it obtains a similar effect to graph neural networks through relation-aware self-attention layers. RATSQL defines several relations for better constructing the schema linking, as shown in Figure 2.6 The relation-aware self-attention layer is based on the transformer architecture, while the GNN and IRNet model is built on the LSTM. RATSQL with BERT achieves the best performance compared to the other models by its excellent design and intensive hyperparameter search. Thus, the RATSQL attracted many follow-up studies based on itself [Shi et al., 2021, Yu et al., 2021].

Chapter 3

Related Works

Text-to-SQL is a task to translate the natural language query (input) written by users into the SQL query (output) automatically. Text-to-SQL is key toward Natural Language Interface to Database (NLIDB). NLIDB has a long history that can be traced back to the 1970s [Warren and Pereira, 1982, Androutsopoulos et al., 1995, Popescu et al., 2004, Li et al., 2006, Iacob et al., 2020]. Most of the early work focuses on single-domain datasets, including ATIS, GeoQuery [Iyer et al., 2017], Restaurants [Tang and Mooney, 2000, Ana-Maria Popescu et al., 2003, Giordani and Moschitti, 2012], Scholar [Iyer et al., 2017], Academic [Li and Jagadish, 2014], Yelp and IMDB [Yaghmazadeh et al., 2017] and so on. Finegan-Dollak et al. [2018] shows some models dealing with specific databases that only learn to match semantic parsing results.

In the remainder of this chapter, we first review the semantic parsing papers. We then discuss the relevant datasets and literature in text-to-SQL.

3.1 Semantic Parsing

The text-to-SQL is related to the general topic of semantic parsing [Kamath and Das, 2018]. Semantic parsing is a task to convert a natural language description to a logical form, where the logical form can be SQL, programming language and etc. Applications of semantic parsing include text-to-SQL [Zhong et al., 2017, Yu et al., 2018b], question answering [Berant et al., 2013, Jia and Liang, 2016], machine translation [Andreas et al., 2013], ontology induction [Poon and Domingos, 2010] and code generation [Rabinovich et al., 2017, Yin and Neubig, 2017b]. The text-to-SQL research has always benefited from the whole semantic parsing community. For example, some models [Dong and Lapata, 2018, Xie et al., 2022] can handle several semantic parsing tasks, including the text-to-SQL.

Early semantic parsing focus on rule-based methods. For example, Woods [1973] proposes LUNAR, a syntactic parser, which generates database query language based on a set of semantic interpretation rules. Most rule-based research was conducted before the 1990s, and a detailed review can be found from [Androutsopoulos et al., 1995]. Although rule-based methods [Templeton and Burger, 1983, Hendrix et al., 1978] streamline the flow for a given task, the inclusion of semantic knowledge and expert-design makes it difficult to adapt to unseen domains.

After the 1990s, researchers began to pay attention to the statistical method. These statistical learning approaches [Zettlemoyer and Collins, 2005, 2007, Kwiatkowksi et al., 2010] are fully supervised using annotated pairs of natural language sentences and logical form. The well-known CHILL [Zelle and Mooney, 1996b] model is trained on a corpus comprising sentences paired with database queries and can map subsequent sentences to executable queries. Thompson [2003] further improve on this method by learning both lexica phrases and meaning representations. However, these statistical methods require complex annotation and can work only in a single domain.

At a similar time, there have been many different approaches proposed for semantic parsing. Poon and Domingos [2009] introduce the first unsupervised method to learn a semantic parser based on Markov logic. Their method successfully extracts a knowledge base from GENIA [Kim et al., 2003] biomedical corpus and can answer questions based on it. Wong and Mooney [2006] employ statistical machine translation approaches to semantic parsing and argue that a parsing task can be viewed as a syntax-based translation task. This model achieves good performance and is more robust to word order, compared with existing learning methods under a similar amount of supervision.

More recently, with the advancement of deep learning, the encoder-decoder structure has become the default solution for semantic parsing. Dong and Lapata [2016] present an attention-enhanced encoder-decoder model for converting natural language (NL) to logical form tasks without high-quality lexicons and manually-built templates. This model encodes input NL utterances into vectors and then uses vectors to generate their logical forms, which follow the seq2seq pipeline. Similarly, Iyer et al. [2017] present an encoder-decoder model with global attention to improving a text-to-SQL semantic parser based on user feedback. Its binary user feedback can improve parser accuracy over time. Yin and Neubig [2017a] introduce a grammar decoder for NL to Python code generation, where this grammar decoder has been used in complex text-to-SQL generation models [Wang et al., 2020]. Dong and Lapata [2018] propose a structure-aware neural model decomposing the semantic parsing process into two steps: (1) generating a rough sketch from NL; (2) generating a final output from both NL and the rough sketch. This model employs two encoder-decoder for different steps and can generate SQL, source code, and logical form from NL. The rough sketch here can be considered an intermediate representation.

Besides text-to-SQL models, some other semantic parsing models also use intermediate representation (IR). For example, Cheng et al. [2017b] introduce a neural semantic parser converting NL descriptions to IR in the form of predicate-argument structures, which are subsequently mapped to target domains. By observing the IR generated by the parser, you can gain insight into what the model has learned Herzig et al. [2021b] study the impact of intermediate representations on compositional generalization in semantic parsing models without changing the model architecture. Experiments show that their proposed IR improves the model robustness against compositional generalization.

3.2 Question Answering

Question Answering (QA) [Calijorne Soares and Parreiras, 2020, Wang, 2022] is an important natural language processing (NLP) task, aiming to generate a corresponding answer to a given question. Text-to-SQL is considered as a table or database based QA, where the answer is a SQL query or the data queried from a table or database. For example, search for the keyword tableQA and you will find many papers about text-to-SQL [Cho et al., 2018, Sun et al., 2020, Jin et al., 2022]. To now, there are several tableQA datasets have been proposed, such as WikiTableQuestions [Pasupat and Liang, 2015], MLB [Cho et al., 2018], and TabMCQ [Jauhar et al., 2016]. In addition to pure tableQA, research on multi-modal QA over text, tables, and images is starting to draw attention from the community [Talmor et al., 2021].

Although the tableQA is related to the text-to-SQL, they are different. As the discussion about the text-to-SQL assumption in Chapter 1, the input of the text-to-SQL must be directly related to SQL instructions, not any question. Therefore, some text-to-SQL input can be in non-problem form, such as: 'Give the name of students.'. To study the input difference between text-to-SQL and QA, we analyze the text-to-SQL corpus Spider [Yu et al., 2018b] and the QA corpus NQ [Kwiatkowski et al., 2019]. NQ denotes Natural Questions and contains 323K questions from real users, while Spider contains 10K and is introduced in Chapter 3.4.1. We first parse the NL questions/descriptions and compare their part-of-speech tag and dependency tree. We found that only there are only 29% Spider NL descriptions that can find a similar parse tree from parsed NQ questions. Although Spider is annotated manually, it is still quite different from the real-world QA question. In the Spider NL descriptions that can not find a similar parse tree from the NQ, most of them are non-questions or too complex.

3.3 Natural Language Interface to Database

The text-to-SQL semantic parsing task, also known as Natural Language Interfaces to Databases (NLIDB), has attracted much attention from the research community since the 1970s [Warren and Pereira, 1982, Androutsopoulos et al., 1995, Popescu et al., 2004, Li et al., 2006, Xu et al., 2017, Yu et al., Dong and Lapata, 2018, Iacob et al., 2020]. The research history and technical route of text-to-SQL almost overlap with semantic parsing. Many semantic parsing methods designed for non text-to-SQL scenarios can also be applied to text-to-SQL. For example, the grammar decoder [Yin and Neubig, 2017a] proposed to generate code was later used to generate SQL [Wang et al., 2020]. For the development history of semantic parsing, please refer to Chapter 3.1.

Text-to-SQL [Qin et al., 2022] is to convert an NL question under the database schema to its corresponding SQL, where the input includes both the NL question and database schema while the output is the SQL query. To learn more about the text-to-SQL task, we can start with the task formulation, evaluation method, methodology, and its corpora. In particular, Chapter 2.1.2 gives a formal task formulation, and Chapter 2.1.3 describes the exact set match and execution match metrics widely used for evaluating the text-to-SQL models. As for the methodology, we discuss it from two perspectives, the model paradigms in Chapter 3.5 and the key modules in Chapter 2.3. In the remainder of this chapter, we review the text-to-SQL corpora and research progress on robust text-to-SQL.

3.4 Text-to-SQL Corpora

Text-to-SQL corpora are essential for learning and evaluating the text-to-SQL parsers. In the following, the text-to-SQL corpora are categorized into either single-turn or multi-turn corpora. Table 3.1 roughly compares these corpora. Since this thesis focuses on the single-turn cross-domain text-to-SQL problem, we discuss it more.

3.4.1 Single-Turn Text-to-SQL Corpora

GeoQuery [Zelle and Mooney, 1996a]. The GeoQuery comprises 880 NL questions issued to a database of US geographical facts (denoted as Geobase), originally in Prolog language. Ana-Maria Popescu et al. [2003] constructed a relational database for GeoQuery together with SQL queries for a subset of 700 questions. Afterwards, the remaining NL questions are further annotated by Iyer et al. [2017].

Dataset	Category	Cross-domain	@Question	@DB	@Domain
GeoQuery	Single-Turn		880	1	1
IMDB	Single-Turn		128	1	1
YELP	Single-Turn		196	1	1
MAS	Single-Turn		131	1	1
WikiSQL	Single-Turn	✓	80654	26521	
Spider	Single-Turn	✓	10181	200	138
Cspider	Single-Turn	✓	10181	200	138
Spider-SSP	Single-Turn				
SparC	Multi-Turn	✓	12726	200	138
CoSQL	Multi-Turn	✓	15598	200	138

Table 3.1: The representative text-to-SQL datasets. DB stands for database. @ denotes the number of the corresponding units.

IMDB, YELP, MAS. The three corpora are proposed in the same paper [Yaghmazadeh et al., 2017]. Each corpus contains one domain(database) containing multi tables. They all contain several tables in each database. There are a total of 131 NL queries in IMDB, 128 in YELP, and 196 in MAS, respectively. The data volume of these three databases is pretty large, all exceeding 1GB.

WikiSQL [Zhong et al., 2017]. The WikiSQL is the first large-scale cross-domain text-to-SQL dataset. WikiSQL contains 80,654 hand-crafted NL question and SQL query pairs along with the corresponding SQL tables. The WikiSQL dataset is more challenging than previous single-domain corpora since the text-to-SQL parsers should generalize to an unseen domain. However, the SQL complexity in WikiSQL is limited: its SQL queries only cover a single *SELECT* column and aggregation, together with relatively simple selection predicates in the *WHERE* clauses, thus lacking in terms of complex SQL queries.

Spider [Yu et al., 2018b]. The Spider dataset is a large-scale cross-domain text-to-SQL benchmark with complex SQL queries. Spider contains 10,181 NL descriptions and 5,693 unique corresponding SQL queries belonging to 138 different domains. The Spider dataset is split into 7,000 examples for training, 1,034 for development, and 2,147 for testing. The databases in training, development, and testing are different, so it also requires models generalizing to the unseen domains. Besides, the SQL queries in the Spider dataset can be divided into four different levels of difficulty: easy, medium, hard, and extra hard. Experiments on Spider have shown that previous models designed for WikiSQL suffered a significant performance drop. This thesis focuses on the Spider benchmark because relatively high generation accuracy has already been achieved for the WikiSQL benchmark, and the SQL structures in Spider cover

WikiSQL vs. Spider	
Similarities	Differences
(1) Both are cross-domain settings without domain knowledge. (2) There are few synonym substitutions in both schema items.	(1) SQL queries in Spider are more complex than that in WikiSQL. (2) Each Spider database contains multi tables, but WikiSQL is not.

Table 3.2: Overall comparison between WikiSQL and Spider

all SQL structures in WikiSQL.

CSpider [**Min et al., 2019**]. The CSpider dataset is a Chinese version of Spider by translating the original English NL utterances into Chinese. Consistent with Spider, CSpider contains the question-SQL pairs as in the Spider dataset.

Spider-SSP [**Shaw et al., 2021b**]. The Spider-SSP is a new train and test split of the Spider dataset based on the Target Maximum Compound Divergence (TMCD) method. The Spider-SSP can evaluate the compositional generalization ability of text-to-SQL models. Spider-SSP consists of 3,282 training instances and 1,094 testing instances. Unlike the Spider, the databases in training and testing are shared, so it does not require models generalizing to the unseen domains.

Comparison Between WikiSQL and Spider

This thesis focuses on the Spider text-to-SQL benchmark because it is more challenging than other datasets and it contains several variants for further studies, such as CSpider and Spider-SSP. At present, the Spider and WikiSQL are the two most popular datasets in the text-to-SQL community. Table 3.2 briefly compares their similarities and differences.

Differences: The most significant difference between WikiSQL and Spider is that SQL queries in Spider are more complex than in WikiSQL. Table 3.3 presents a complex SQL example from Spider, in which the question seems conceptually simple but involves several different pieces of database tables and SQL clauses.

Additionally, the Spider database contains several tables while there is only one table in the WikiSQL database. The presence of multiple tables introduces column and table name disambiguation problems to Spider, whereas none exist in WikiSQL. For example, suppose that the tables ‘student’, ‘course’, and ‘studentship’ all contain a ‘student ID’ column. You would need to choose one ‘student ID’ column from these tables when the question is ‘Show the student ID

Question:	What airports don't have departing or arriving flights?

SQL:	**SELECT** AirportName **FROM** Airports
	WHERE AirportCode **NOT IN** (
	SELECT SourceAirport **FROM** Flights **UNION**
	SELECT DestAirport **FROM** Flights)

Table 3.3: A complex nested SQL with set operator

who choose math'. Multiple tables in Spider also cause the number of columns to be dozens of times more massive than WikiSQL, which increases the difficulty of choosing the correct column.

Similarities: Although WikiSQL and Spider are cross-domain settings, most SQL queries do not need domain knowledge during generation. The domain knowledge usually is a consensus that only exists in a specific field and will not be clearly stated in the question. For example, in a restaurant booking scenario where domain knowledge is needed, the question requires a 'good restaurant', which means its rating star must be higher than 3.5. More domain knowledge details can be found in Appendix A.

In addition, most sentences use schema annotation words instead of synonyms, allowing the model to locate the schema items through exact word matching. For example, in Table 3.3, the question rarely uses 'airplane' or other synonyms to replace 'flight', given that the schema column word is 'flight'.

3.4.2 Multi-Turn Text-to-SQL Corpora

SParC [Yu et al., 2019b]. The SParC is a large-scale cross-domain context-dependent text-to-SQL dataset built on the Spider, using the same databases as the Spider. Each SParC question sequence is based on a Spider question by asking inter-related questions. After obtaining the sequential questions, annotators give the SQL query to each question. In total, the SParC contains 4k+ question sequences including 12k+ question-SQL pairs.

CoSQL [Yu et al., 2019a]. The CoSQL dataset is the first large-scale cross-domain conversational text-to-SQL dataset, containing about 3k dialogues including 30k+ turns and 10k+ corresponding SQL queries. The CoSQL follows the Wizard-of-Oz [Budzianowski et al., 2018] settings, recruiting annotators who act as DB users and SQL experts respectively to simulate a DB query scenario. Experiments show that the baseline model performance on CoSQL suggests plenty of space for improvement.

3.5 The Paradigms of Text-to-SQL

There are several text-to-SQL model paradigms, such as (1) using rule-based methods to generate the SQL query [Androutsopoulos et al., 1995], and (2) using a re-ranking strategy [Bogin et al., 2019b] to select the best SQL query. However, the performance of the first method is relatively low, and the second approach cannot run independently. Therefore, we select representative independent high-performance text-to-SQL models, ignoring their removable modules that improve performance and divide these models into two paradigms, as shown in Figure 3.1.

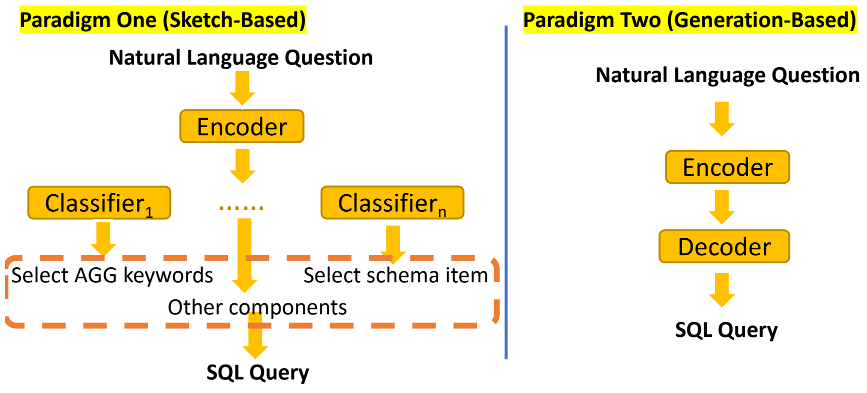


Figure 3.1: General structure of two text-to-SQL paradigms.

3.5.1 Paradigm One (Sketch-Based Method)

The sketch-based methods predefine a class of classifiers to determine which schema item or SQL keywords to be generated, as shown in Figure 3.1. We briefly introduce the sketch-based methods for SQL keyword generation in Chapter 2.3.1. Actually, these methods are also used for selecting the schema items.

In WikiSQL, because the dataset only contains simple SQL, most models decompose the SQL synthesis into several independent classification sub-tasks. Each sub-task employs an independent classifier, taking the entire sentence as input. For example, one classifier would be used to determine which column is the column in *SELECT* clause, and another separate classifier to determine which aggregation function is correct. These models include: SQLNet [Xu et al., 2017], TypeSQL [Yu et al.], SQLova [Wonseok Hwang, Jinyeung Yim, Seunghyun Park, 2019], HydraNet [Lyu et al., 2020], X-SQL [He et al., 2019], Coarse2Fine [Dong and Lapata, 2018] and etc.

However, this paradigm is only effective in simple SQL generation problems because it requires too many classifiers for complex SQL, leading to a complex model structure and relatively poor performance [Yu et al., 2018b,a]. For example, the SQLNet and TypeSQL models designed for WikiSQL have been transferred to Spider; however, their performance has dropped significantly. SyntaxSQLNet [Yu et al., 2018a] is the first model designed for Spider and, based on a similar idea, uses independent modules to predict different clauses. However, its performance is lower than the later models belonging to Paradigm two [Guo et al., 2019, Bogin et al., 2019a].

3.5.2 Paradigm Two (Generation-Based Method)

Compared to Paradigm One, the structure of Paradigm Two is pretty concise, relying on a single decoder to generate the SQL query, as shown in Figure 3.1. The generation-based method follows seq2seq modeling (introduced in Chapter 2.2.2), generating target SQL during the decoding process. However, the early seq2seq model cannot work well in text-to-SQL [Yu et al., 2018b]. To solve this problem, [Xiao et al., 2016, Cheng et al., 2017a, Yin and Neubig, 2017a] propose grammar decoder to replace standard decoder. Experiments show that the generation-based method with grammar decoder consistently outperforms the sketch-based method in the Spider complex text-to-SQL generation benchmark [Yu et al., 2018b, Wang et al., 2020, Guo et al., 2019, Bogin et al., 2019a].

However, the grammar decoder is not perfect in concurrently generating the SQL keywords and schema items. For example, as shown in Table 3.4, we tested the top models (RATSQL [Wang et al., 2020], IRNet [Guo et al., 2019], and GNN [Bogin et al., 2019a]) in the Spider leaderboard and all these models tended to generate wrong predictions, as shown in the table. Because there is no strong interaction between generating SQL structure (generating the error 'avg' function) and filling the schema item (filling the 'average' column). SQL structure generation depends on sentence analysis while filling the schema items depends on the similarity between schema items and sentence tokens. This phenomenon will prevent the generation of a correct SQL when there is a column named 'average' or 'max' or 'min' in the schema [Gan et al., 2021b].

Question:	What is the average salary.
Gold SQL:	**SELECT** average **FROM** salary
Wrong prediction:	**SELECT** avg(average) **FROM** salary

Table 3.4: A common prediction error

The IE-SQL [Ma et al., 2020] model brings hope for the grammar decoder problem. IE-SQL is an information extraction-based text-to-SQL method that tackles tasks via sequence labeling, relation extraction, and text matching. IE-SQL first automatically labels questions by analyzing their corresponding SQL, then trains a neural model to learn how to label a question without an SQL. Finally, IE-SQL can synthesize an SQL from the question labels deterministically. Figure 3.2 illustrates the two generation steps in the IE-SQL. IE-SQL is also a generation-based model employing an encoder-decoder to generate the question labels, where the role of the labels is similar to that of intermediate representation.

Although this approach seems to avoid the problem in Table 3.4, generating the correct annotation and then synthesizing an SQL from the question label for Spider requires much more work than WikiSQL might because Spider's sentences and SQL queries are much more complicated than WikiSQL. The same name column, which has not appeared in WikiSQL, also restricts applying this method directly for Spider. While it is difficult to use IE-SQL on Spider, it may work well by generating labels first and then training a grammar decoder model with labels.

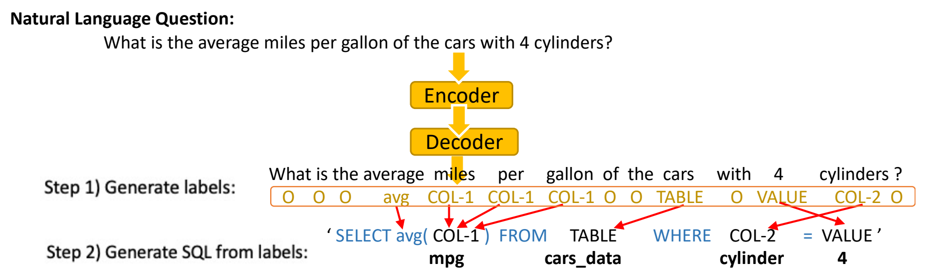

Natural Language Question:
What is the average miles per gallon of the cars with 4 cylinders?
Encoder
Decoder
Step 1) Generate labels:
What is the average miles per gallon of the cars with 4 cylinders ?
O O O avg COL-1 COL-1 COL-1 O O TABLE O VALUE COL-2 O
Step 2) Generate SQL from labels:
' SELECT avg(COL-1) FROM TABLE WHERE COL-2 = VALUE '
mpg
cars_data
cylinder
4

Figure 3.2: IE-SQL [Ma et al., 2020] model generates the question labels and then synthesize the target SQL from the labels.

3.6 Robust Text-to-SQL

We discuss robust text-to-SQL from four aspects data, SQL keyword generation, schema linking, and compositional generalization.

3.6.1 Data

Existing works on improving the robustness of the text-to-SQL model are mainly through adversarial training and data augmentation. Xiong and Sun [2019] propose an AugmentGAN model to augment more data for the target domain. Li et al. [2019] use a sentence shuffling method to augment the single domain GeoQuery and Restaurants dataset. Radhakrishnan et al. [2020] augment the WikiSQL [Zhong et al., 2017] dataset with synthetic search-style questions to improve the robustness of short, colloquial input. Recent work demonstrates that adversarial training with augmented data can improve performance and robustness of text-to-SQL models [Zhu et al., 2020]. Zeng et al. [2020] introduce a Spider$_{\text{UTran}}$ dataset that includes original Spider [Yu et al., 2018b] examples and some untranslatable questions examples. Zeng et al. [2020] also study the robustness of the text-to-SQL model when the user inputs an untranslatable NL question.

3.6.2 SQL Keyword Generation

As discussed in Chapter 3.5.2, Gan et al. [2021b] find that the grammar decoder tends to generate incorrect SQL keywords when there is a column named 'average' or 'max' or 'min' in the schema. Although the method in IE-SQL can solve this problem to a certain extent, the current IE-SQL is not suitable for the complex text-to-SQL task. In addition, Gan et al. [2021b] also introduces other types of domain knowledge that cause SQL keyword generation errors. For example, The model cannot understand that there should be different order keywords for age and birthday when sorting from old to young, e.g., DESC used for age and ASC used for birthday. Experiments show that the introduction of external knowledge can alleviate these problems.

3.6.3 Schema Linking

Schema linking, introduced in Chapter 2.3.2, is a key module for models to generate correct schema items. So if the schema linking is destroyed, the model robustness will be affected. Guo et al. [2019] and Wang et al. [2020] conducted an ablation study on exact match based schema linking (EMSL), respectively, and the results showed that removing the EMSL would lead to the greatest decrease in model performance. These studies have influenced many follow-up works using EMSL [Cai et al., 2021, Xu et al., 2021, Lei et al., 2020, Yu et al., 2021, Shi et al., 2021]. However, we found that once the model uses EMSL, it becomes reliant on it, and pretrained language models can replace EMSL and make the model more robust, see Chapter 6.

In addition to discussing schema linking in the paper as part of the model [Guo et al., 2019, Bogin et al., 2019a, Wang et al., 2020, Chen et al., 2020a, Cao et al., 2021], some works have focused on the schema linking. Lei et al. [2020] demonstrated that more accurate schema linking conclusively leads to better text-to-SQL parsing performance. To support further schema linking studies, Lei et al. [2020] and Taniguchi et al. [2021] invested human resources into annotating schema linking corpus.

3.6.4 Compositional Generalization

Compositional generalization is the ability to generalize to novel combinations of the components observed during training. Compositional generalization is a basic capability of human beings, but neural network models have been proven to lack these capabilities. Figure 1.3 gives a compositional generalization example in text-to-SQL, discussed in Robust Text-to-SQL Parsing of Chapter 1.

Compositional generalization for semantic parsing has garnered a great deal of attention [Finegan-Dollak et al., 2018, Oren et al., 2020, Furrer et al., 2020, Conklin et al., 2021]. Most prior works on text-to-SQL tasks focus on the cross-domain generalization, which mainly assess how the models generalize the domain knowledge to new database schemas [Suhr et al., 2020, Gan et al., 2021b]. Oren et al. [2020] study the compositional split in several text-to-SQL dataset Finegan-Dollak et al. [2018], and find some factors to improve generalization performance. Besides, Shaw et al. [2021b] introduced Target Maximum Compound Divergence (TMCD) splits for studying compositional generalization in semantic parsing, where they aimed to maximize the divergence of SQL compounds, that are one or multi SQL clauses, between the training and test sets. The TMCD split only ensures that the NL question word atom and its corresponding SQL compounds appear in the training set and do not care about its semantic form. The TMCD split also requires that the SQL queries in the test set be as different as possible from the training set. For example, the TMCD split requires model learning '*Give me the name of the student who is the oldest*' can predict the '*Give me the name of the oldest student*' because the SQL compounds in testing appear in training.

Besides the research devoted to text-to-SQL, the research on the compositional generalization of semantic parsing also brings inspiration. Furrer et al. [2020] investigate state-of-the-art techniques and architectures to assess their effectiveness in improving compositional generalization in semantic parsing tasks. Yin et al. [2021] described a span-level supervised attention loss that would improve compositional generalization in semantic parsers. Herzig and Berant [2021] proposed SpanBasedSP, a parser that could predict a span tree over an input utterance, dramatically improving performance on splits that required compositional generalization. Chen et al. [2020b] proposed the neural-symbolic stack machine, which could integrate a symbolic stack machine into a seq2seq generation framework, learning a neural network as the controller to operate the machine.

Chapter 4

Natural SQL (NatSQL)

In Chapter 3, we reviewed different SQL IRs and how these IRs bridge the mismatch between NL and SQL [Guo et al., 2019]. However, previous IRs are too complicated or have limited coverage of SQL structures. Besides, although the existing IRs eliminate part of the mismatch between intent expressed in NL and the implementation details in SQL, some mismatches can be further eliminated by improving the IR.

In this chapter, we present *Natural SQL (NatSQL)*, a new intermediate representation that offers simplified queries over other IRs, while preserving a high coverage of SQL structures. More importantly, NatSQL further eliminates the mismatch between NL and SQL, and can easily support executable SQL generation. Figure 1.4 presents a sample comparison between NatSQL and other IRs. We observe that there is a mismatch between the NL word `'and'` and the *INTERSECT* SQL keyword, since in another similar question shown in Figure 4.6, the `'and'` no longer corresponds to the *INTERSECT* keyword. To translate the NL question into a corresponding query, previous IRs need the models to distinguish whether the word `'and'` corresponds to *INTERSECT*, this is not required for NatSQL. Among all IRs, NatSQL provides the simplest and shortest translation, while the NatSQL structure also aligns best with the NL question.

NatSQL preserves the core functionalities of SQL, while simplifying the queries as follows: (1) dispensing with operators and keywords such as *GROUP BY, HAVING, FROM, JOIN ON*, which are usually hard to find counterparts for in the text descriptions; (2) removing the need for nested subqueries and set operators, using only one *SELECT* clause in NatSQL; and (3) making schema linking easier by reducing the required number of schema items that are normally not mentioned in the NL question. The design of NatSQL easily enables executable SQL generation, which is not naturally supported by other IRs.

We compare NatSQL with SQL and other IRs by incorporating them into existing open-source neural network models that achieve competitive performance on Spider. Our experiments show that NatSQL boosts the performance of these existing models, and outperforms both SQL and other IRs. In particular, equipping RAT-SQL+GAP with NatSQL achieves a new state-of-the-art execution accuracy on the Spider benchmark. These results suggest that to improve the ability of text-to-SQL models to understand and reason about the NL descriptions, designing IRs to better reveal the correspondence between natural language (NL) and query languages is a promising direction.

This chapter is based on [Gan et al., 2021c]. Our contributions in this chapter are as follows:

- We propose a SQL IR named NatSQL to bridge the gap between NL and SQL better.
- We modify several models to fit the NatSQL and attain performance improvement.
- We evaluate the performance of IRs on the different text-to-SQL models. Experiments show that NatSQL outperforms other IRs.

4.1 Two Steps Toward Text-to-SQL with IR

The ML-based text-to-SQL models mostly divide the problem into two subtasks: generating SQL keywords (blue character in Figure 1.4) and filling the schema items (black character in Figure 1.4), also named schema linking. When the text-to-SQL models use IR, the change from text-to-SQL to text-to-IR can be done without modifying the model structure. For related discussion, please refer to Chapter 2.3.3. We investigate how we can design an IR to improve both SQL keyword generation and schema item generation.

4.1.1 Generating SQL Keywords

Neural text-to-SQL models usually generate the SQL keywords according to the similarity linking scores between the hidden state from the question and the production rule embeddings. For example, in Figure 1.4, we conjecture a good text-to-SQL model should be able to give a higher linking score between the word '*less*' and the SQL '<' keyword.

However, SQL is designed for effectively querying relational databases, not for representing the meaning of NL questions. Hence, there inevitably exists a mismatch between intents expressed in natural language and the implementation details in SQL [Guo et al., 2019]. For example, in Figure 1.4, the *GROUP BY* and *JOIN ON* clauses are not mentioned in the question. One solution is to use an IR to remove the SQL clauses that are hard to predict. Experiments show that the SemQL IR can improve the accuracy of previous models [Guo et al., 2019].

4.1.2 Generating Schema Items

Text-to-SQL models usually generate the schema items according to the similarity linking scores between tokens in the question and database schemas. Intuitively, a model is supposed to predict higher scores to schema items that are mentioned in the question. To achieve this goal, some existing neural networks implement a schema linking mechanism, by recognizing the tables and columns mentioned in a question [Guo et al., 2019, Bogin et al., 2019a, Wang et al., 2020].

Schema linking is essential for text-to-SQL tasks. As shown in the ablation study of IRNet [Guo et al., 2019] and RAT-SQL [Wang et al., 2020], removing the schema linking results in a dramatic decrease in performance. The importance of schema linking raises a question about generating schema items not mentioned in the question. Some models use graph neural networks to find these unmentioned schema items, and some models delete unmentioned schema items based on the IR; e.g., in Figure 1.4, the IRs remove the *JOIN ON* and *GROUP BY* clauses with the unmentioned schema items.

4.2 NatSQL

In this section, we present the detail of NatSQL. NatSQL is an SQL intermediate representation that simplifies the SQL structure and makes schema linking easier.

4.2.1 Assumption

We design NatSQL on an assumption: when users use natural language to query the database, the NL descriptions are: the required data and the query conditions. Based on this assumption, we think it is possible to design an SQL IR that only needs two different clauses corresponding to the required data and query conditions, respectively. We found that the required data only corresponds to the *SELECT* clause of SQL, and the remaining SQL clauses correspond to query conditions. In particular, *WHERE* is obviously the simplest and most important clause among these conditional clauses. Therefore, we started to study how to combine the functions of all other clauses into the *WHERE* clause and found it can be done. However, when incorporating *ORDER BY* into the *WHERE*

NatSQL	=	*SELECT* , Column , { ‘,’ Column } , [*WHERE* W_Cond] , [*ORDER BY* Order_By] ;
Column	=	Agg_Col | Table_Col ;
Agg_Col	=	Agg_Fun , ‘(’ Table_Col , ‘)’ ;
Agg_Fun	=	‘*avg*’ | ‘*count*’ | ‘*max*’ | ‘*min*’ | ‘*sum*’ ;
Table_Col	=	TABLE_NAME , ‘.’ , COLUMN_NAME | TABLE_NAME , ‘.’ , ∗ ;
W_Cond	=	[Conjunct], Condition , { Conjunct Condition } ;
Condition	=	Cond_L , W_Oper , Cond_R , [‘and’ , NUMBER] ;
Conjunct	=	‘*and*’ | ‘*or*’ | ‘*except*’ | ‘*intersect*’ | ‘*union*’ | ‘*sub*’ ;
W_Oper	=	‘*between*’ | ‘=’ | ‘>’ | ‘<’ | ‘>=’ | ‘<=’ | ‘! =’ | ‘*in*’ | ‘*like*’ | ‘*is*’ | ‘*exists*’ | ‘*not in*’ | ‘*not like*’ | ‘*not between*’ | ‘*is not*’ | ‘*join*’ ;
Cond_R	=	NUMBER | STRING | Column ;
Cond_L	=	Column | “@” ;
Order_By	=	Column , [*DESC* | *ASC*] , [*LIMIT* , NUMBER]

Table 4.1: The main grammar of NatSQL. Here we highlight the differences of production rules from SQL.

clause, we found it to be not elegant, so we decided to retain two conditional clauses: *WHERE* and *ORDER BY*. Finally, we propose the NatSQL, an SQL IR with only the *SELECT*, *WHERE*, and *ORDER BY* clauses. We believe that SQL IR should not seek to cover all forms of SQL in the text-to-SQL scenarios, otherwise it is better to use SQL directly. In a text-to-SQL process, we are happy with the NatSQL converted to an equivalent SQL that is not the same as the target one.

4.2.2 Overview

Table 4.1 presents the grammar specification of NatSQL defined in extended Backus-Naur form (EBNF) [Scowen, 1993]. The lowercase symbols in a single quotation and the capital symbols are terminal, while other symbols are nonterminal. The curly bracket represents that symbols inside it appear zero to multi times. The square bracket denotes that symbols inside it appear zero or one time. The symbols separated by the ‘|’ are juxtaposed, and any one of them can substitute the symbol on the left of the ‘=’ symbol. To better understand the production rules, Figure 4.1 presents an example of NatSQL with its grammar symbols.

NatSQL only retains the *SELECT*, *WHERE* and *ORDER BY* clauses from SQL, dispensing with other clauses such as *GROUP BY, HAVING, FROM, JOIN ON*, set operators and subqueries. Symbols in capital italics or in a single quotation are keywords of SQL and NatSQL, and other capital symbols represent special meanings, where 'TABLE_NAME' and 'COLUMN_NAME' are defined for databases, and 'NUMBER' and 'STRING' represent the data types.

Except for the deleted clauses, the differences between NatSQL and SQL are underlined in Table 4.1. NatSQL implements the function of the deleted clauses by adding new keywords and allowing *conjunct* to appear before the WHERE condition. In terms of language format, NatSQL does not add new clauses, and can retain deleted clauses as needed.

The main design principle of NatSQL is to simplify the structure of SQL and bring its grammar closer to natural language. Considering the example in Figure 1.4, the set operator '*INTERSECT*', used to combine *SELECT* statements, is never mentioned in the question. *INTERSECT* is introduced in SQL to allow the combination of the results of multiple functions. Such implementation details, however, are rarely considered by end users and therefore rarely mentioned in questions [Guo et al., 2019].

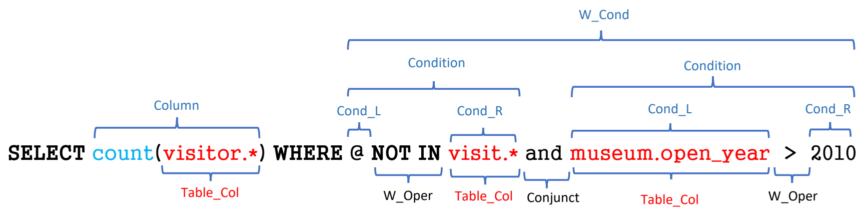


Figure 4.1: An example of NatSQL corresponding to the grammar symbols in Table 4.1

4.2.3 Overall Comparison

Starting from SyntaxSQLNet [Yu et al., 2018a], several types of IR have been developed for text-to-SQL models on the Spider dataset. The main limitation of SyntaxSQLNet is that it removes the *FROM* and *JOIN ON* clauses, which may result in the failure to find the correct table when converted to SQL. For example, in Figure 1.4, SyntaxSQLNet IR misses the *inventory* table, thus it cannot generate the correct *JOIN ON* clause that appears in the original SQL. The IR for RAT-SQL [Wang et al., 2020] is mostly close to SQL, and it avoids missing tables since it only removes the *JOIN ON* clause from SQL. Zhong et al. [2020b] and Lee [2019] also utilize an IR that is similar to the IR in RAT-SQL and SyntaxSQLNet.

Guo et al. [2019] introduced SemQL, an intermediate language, to facilitate SQL prediction. As with NatSQL, SemQL removes the keywords *FROM, JOIN ON, GROUP BY, HAVING* from SQL. Although SemQL and NatSQL remove both *FROM* and *JOIN ON* clauses, SemQL and NatSQL avoid missing a table by moving the table into the '*' column. NatSQL improves on SemQL in the following ways:
(1) Compatible with a wider range of SQL queries than SemQL.
(2) Simplify the structure of queries with set operators, i.e., *INTERSECT*, *UNION*, and *EXCEPT*, denoted as *IUE* hereafter.
(3) Eliminate nested subqueries.
(4) Reduce the number of schema items to predict.
(5) NatSQL uses the same keywords and syntax as SQL, which makes it easier to read and expand than SemQL.

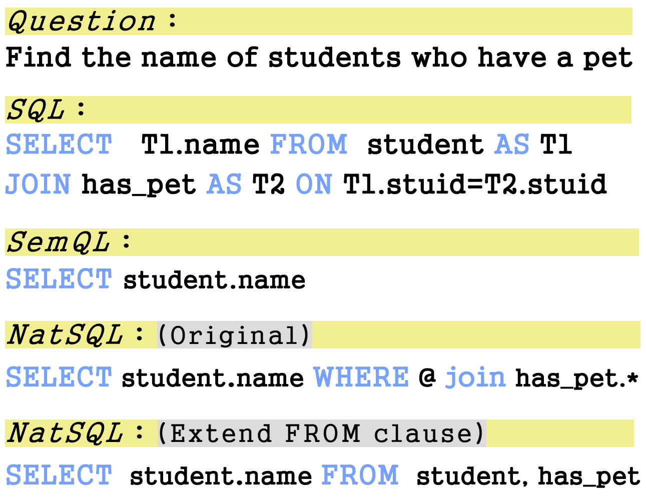
There are four examples in Figure 1.4, 4.2, 4.3 and 4.4 demonstrating the differences between SQL, SemQL, and NatSQL statements representing the same natural language question.

4.2.4 Scalability of NatSQL

We take an SQL query with multiple tables as an example. In Figure 4.2, since the SemQL misses the *has_pet* table, SemQL cannot be converted to the target SQL, indicating that SemQL is not compatible with this type of SQL query. The SyntaxSQLNet IR is also not compatible, but the RAT-SQL IR can convert this query appropriately.

While both SemQL and NatSQL completely remove all *FROM* and *JOIN ON* clauses, NatSQL introduces a new *WHERE* condition operator *join* for these unremovable *JOIN ON* clauses, as shown in Figure 4.2. With this extra *WHERE* condition, NatSQL can be converted to the target SQL. Alternatively, you could use the NatSQL augmented with *FROM* clause version. We recommend the original version since its experimental result is better and the sub-question `'who have a pet'` looks like a *WHERE* condition. We modify this example in Table 4.2 to illustrate why it looks like a *WHERE* condition. Usually, NatSQL does not need the *join* operator for generating *JOIN ON* clause, such as the '*Ques 2*' in Table 4.2, except in cases when it cannot infer the correct *JOIN ON* clause from other clauses.

NatSQL$_\mathbf{G}$. Since each database has different compatibility with SQL, we allow NatSQL to retain the deleted clauses as needed. NatSQL$_\mathbf{G}$ is NatSQL augmented with *GROUP BY*, which improves the compatibility in the SQLite database where the Spider benchmark is built on.



```
Question:
Find the name of students who have a pet

SQL:
SELECT  T1.name FROM  student AS T1
JOIN has_pet AS T2 ON T1.stuid=T2.stuid

SemQL:
SELECT student.name

NatSQL: (Original)
SELECT student.name WHERE @ join has_pet.*

NatSQL: (Extend FROM clause)
SELECT  student.name FROM  student, has_pet
```

Figure 4.2: An example about the scalability and readability of NatSQL.

Ques 1:	Find ... who have a pet.
NatSQL:	... **WHERE** @ *join* has_pet.*
Ques 2:	Find ... who have two pet.
NatSQL:	... **WHERE** *count*(has_pet.*) = 2

Table 4.2: A modified example based on Figure 4.2

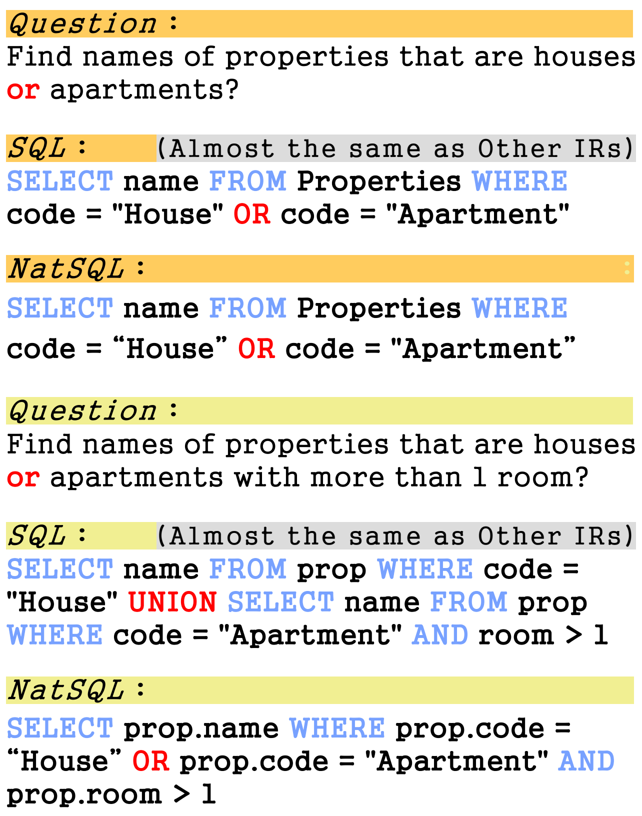
4.2.5 NatSQL for SQL Keyword Generation

By simplifying the set operators and nested subqueries, NatSQL improves text-to-SQL models.

Simplifying Queries with Set Operators

It is typically hard to generate queries with IUE (*INTERSECT*, *UNION*, and *EXCEPT*) set operators for text-to-SQL models, where the corresponding F1 score is usually the lowest among all breakdown metrics on the Spider benchmark [Guo et al., 2019, Bogin et al., 2019a, Wang et al., 2020]. The main reason is that the related questions are generally longer and more complicated, while the mismatch between NL and SQL queries further increases the prediction difficulty, as discussed in Section 4.1.1.

Figure 4.3 compares the SQL queries corresponding to two similar problems. The second question in Figure 4.3 contains an extra condition: `‘more than 1 room’`. This extra condition changes the structure of the entire SQL query. Although IRs have been widely used for complex SQL, enthusiasts of end-to-end models expect the text-to-SQL model to automatically distinguish whether the word



Question :
Find names of properties that are houses or apartments?

SQL : (Almost the same as Other IRs)
```
SELECT name FROM Properties WHERE
code = "House" OR code = "Apartment"
```

NatSQL :
```
SELECT name FROM Properties WHERE
code = "House" OR code = "Apartment"
```

Question :
Find names of properties that are houses or apartments with more than 1 room?

SQL : (Almost the same as Other IRs)
```
SELECT name FROM prop WHERE code =
"House" UNION SELECT name FROM prop
WHERE code = "Apartment" AND room > 1
```

NatSQL :
```
SELECT prop.name WHERE prop.code =
"House" OR prop.code = "Apartment" AND
prop.room > 1
```

Figure 4.3: An example about the mismatch between NL and IUE set operators.

token 'or' in Figure 4.3 corresponds to *UNION* or *OR* keyword. However, most models cannot do that and would generate a *OR* clause for both questions. This example is similar to the comparison between Figure 1.4 and Figure 4.6 discussed in the beginning of this chapter.

NatSQL bridges this gap by unifying them into a simple *OR* operator that will be converted to a *UNION* clause when it cannot concatenate its following conditions. The reasons for the failure to concatenate conditions include: (1) the precedence of the following conditions is higher (e.g., the precedence of *AND* is higher than *OR*); (2) the two conditions cannot be connected, or they are disjoint such as the example in Figure 1.4. The '*count(film_actor.*)>5*' condition cannot be connected with the '*count(inventory.*)<3*' condition because they belong to different tables. Based on the same rules, NatSQL can simplify the SQL with *INTERSECT* (example is shown in Figure 1.4) and *EXCEPT*. As to the case that the set operator itself represents part of a condition, NatSQL allows them to follow the *WHERE* keyword. As illustrated in Table 4.3, this type of SQL is mainly related to the *EXCEPT* operator.

The NatSQL prediction work in Table 4.3 is easier than others. NatSQL here only needs to predict the '*cartoon*' table, instead of predicting the '*cartoon.channel*' column. Predicting a table is easier than predicting a column because the premise of finding the correct column is to find the correct table. Besides, many models incorrectly output '*cartoon.id*' instead of '*cartoon.channel*' because the annotation of '*cartoon.id*' is the same as'*tv_channel.id*' column.

Ques	Find the id of tv channels that do not play any cartoon
SQL	**SELECT** id **FROM** tv_channel **EXCEPT** **SELECT** channel **FROM** cartoon
SemQL	**SELECT** tv_channel.id **EXCEPT** **SELECT** cartoon.channel
NatSQL	**SELECT** tv_channel.id **WHERE except** cartoon.*

Table 4.3: An example of none *WHERE* conditions before the IUE.

In addition to the conditions mentioned above that cannot be concatenated, Table 4.4 present one more example. These two conditions can not concatenate because one *WHERE* condition can not concatenate a *HAVING* condition by a *OR* operator.

Ques	Which film is rented at a fee of 0.99 **or** has less than 3 in the inventory?
SemQL	**SELECT** film.title **WHERE** film.rental_rate = 0.99 **UNION** **SELECT** film.title **WHERE** count(inventory.*)< 3
NatSQL	**SELECT** film.title **WHERE** film.rental_rate = 0.99 **OR** count(inventory.*)< 3

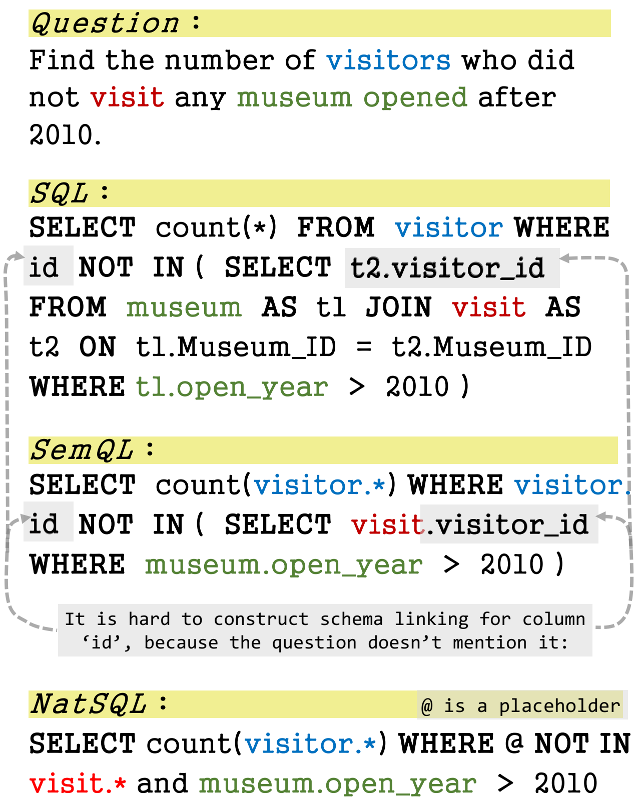
Table 4.4: An example modified from that in Figure 5.

Eliminating Nested Subqueries

Since the subqueries in both NatSQL and SemQL only appear in *WHERE* conditions, only one column in the *SELECT* clause of a subquery is required. NatSQL keeps this *SELECT* column in '*Cond_R*' (right column of *WHERE* conditions) instead of a whole *SELECT* clause. Since this meets the *WHERE* condition format, NatSQL can remove the brackets and subqueries from SQL, as shown in Figure 4.4.

4.2.6 How NatSQL Help Schema Item Generation

NatSQL helps schema item generation by reducing the number of schema items that need to be predicted. For example, in Figure 4.4, without an in-depth



Question:

Find the number of visitors who did not visit any museum opened after 2010.

SQL:

```
SELECT count(*) FROM visitor WHERE
id NOT IN ( SELECT t2.visitor_id
FROM museum AS t1 JOIN visit AS
t2 ON t1.Museum_ID = t2.Museum_ID
WHERE t1.open_year > 2010 )
```

SemQL:

```
SELECT count(visitor.*) WHERE visitor.
id NOT IN ( SELECT visit.visitor_id
WHERE museum.open_year > 2010 )
```

It is hard to construct schema linking for column 'id', because the question doesn't mention it:

NatSQL: @ is a placeholder

```
SELECT count(visitor.*) WHERE @ NOT IN
visit.* and museum.open_year > 2010
```

Figure 4.4: A sample question in Spider dataset with corresponding SQL, SemQL and NatSQL queries.

analysis of the database schema, by looking at the natural language description itself, it is difficult to infer the grey shaded columns in SQL and SemQL (in this example, they are column '*id*' in table '*visitor*' and column '*visitor_id*' in table '*visit*'). We cannot build a schema linking for these columns, even though the schema linking is important to boost performance as discussed in Section 4.1.2.

NatSQL solves this problem by replacing some of the columns with a table only or @, where @ is a placeholder of NatSQL. We can find that all columns of NatSQL in Figure 4.4 are mentioned in the question. Specifically, NatSQL uses @ to replace the '*visitor.id*' and uses '*visit.**' to replace '*visit.visitor_id*'.

@ is a placeholder in NatSQL that only appears in '*Cond_L*', which denotes that we need to infer a column to replace it. The '*' keyword does not appear in the *WHERE* condition without an aggregation function, so NatSQL uses it to represent a table. With this table, we can infer the correct column in the target SQL to replace the @ and '*table.**' according to Algorithm 1.

Algorithm 1 Infer columns to replace the @ and table.* in NatSQL

```
Input: $t\_list$ ▷ All tables before @, which include the table 'visitor' in Figure 4.4
       $table\_r$ ▷ The table next to the @, which is the table 'visit' in Figure 4.4
Output: Two columns to replace the @ and table.*
1: for Every $table$ in $t\_list$ do
2:     if There is foreign key relationship between $table$ and $table\_r$ then
3:         return These two foreign key columns
4: for Every $table$ in $t\_list$ do
5:     if There are columns with the same name in both $table$ and $table\_r$ then
6:         return The same name columns
7: return Their primary keys
```

4.3 Generate SQL From NatSQL

NatSQL is similar to SQL while their *SELECT* and *ORDER BY* clauses are the same. But we still need to generate SQL from NatSQL since the NatSQL cannot be run by database systems.

4.3.1 Generate *HAVING*

NatSQL combine the *HAVING* and *WHERE* condition into the *HAVING* clause because, unlike *HAVING* condition, the SQL *WHERE* condition does not use aggregate functions in its '*Cond_L*'. Therefore, we can restore the *HAVING* conditions by collecting all conditions whose '*Cond_L*' use aggregate functions, while the remaining conditions are SQL *HAVING* conditions.

4.3.2 Generate *GROUP BY*

In most case, the column in *GROUP BY* should appear in the *SELECT* clause which makes the query more reasonable. For example, in the question "Show average salary on each group.", if you only show the average salary without the group information, you have no idea the average salary belongs to which group. NatSQL generate the *GROUP BY* clause by copying the *SELECT* columns without an aggregate function to the *GROUP BY* clause. We need to generate *GROUP BY* clause from NatSQL in one of the following circumstance:

- There are columns with and without aggregate functions that appear in the *SELECT* clause.
- There are *HAVING* conditions in the SQL generated from NatSQL.
- There are columns with aggregate functions that appear in the *ORDER BY* clause.

4.3.3 Generate *FROM* and *JOIN ON*

NatSQL does not support generating all types of *FROM* and *JOIN ON* clause. We discuss them separately.

Supported Types

For the simplest *FROM* followed by a single table without *JOIN ON*, NatSQL can be well compatible. For *JOIN ON*, NatSQL supports the most common type, i.e., use the foreign key relationship to build the *ON* conditions. Since every column in NatSQL contains a table name, including the '$*$' column, we can infer the correct *FROM* and *JOIN ON* clause from the used tables by a heuristic search.

For example, here is a NatSQL: "**SELECT** A.a **WHERE** C.c = 'V'" which contains table A and C in it. We assume that we can search the foreign key relationships from all tables and then find a path starting from A or C to C or A. We may find that column A.b is the foreign key reference to column B.b, and column B.c is the foreign key reference to column C.c. Now, we can infer the path is "**FROM** A **JOIN** B **ON** A.b = B.b **JOIN** C **ON** B.c = C.c". But if we cannot find a path starting from A or C to C or A, we cannot infer the runnable SQL from the previous NatSQL. The solution is that we should also search a list of possible *JOIN ON* relationships that the database developers must manually create.

However, if column A.b and A.bb are both the foreign keys reference to column B.b, there are two available *ON* conditions: "A.b = B.b" and "A.bb = B.b". NatSQL-to-SQL will choose the condition mentioned by the question, i.e., whether the question mentions A.b or A.bb column. If both columns are not mentioned, we can only choose one randomly, which may be wrong. Otherwise, we can move the *ON* condition into the NatSQL *WHERE* condition, which is "**SELECT** A.a **WHERE** C.c = 'V' and A.b = B.b".

Unsupported Types

In addition to the possible incompatibility caused by multiple feasible foreign key relationships for *ON* condition, NatSQL is not compatible with the following three circumstances:

- Self connection, such as: *FROM A JOIN A ON* $A.a = A.b$.
- The operator in *ON* condition is not "=", such as: *FROM A JOIN B ON* $A.a > B.b$.
- subquery substitute for a table name after *FROM* or *JOIN*.

4.3.4 Generate Subquery

NatSQL is not compatible with subqueries appearing in the *FROM* clause but is compatible with subqueries appearing in *WHERE* and *HAVING* conditions. Since NatSQL combines the *WHERE* and *HAVING* conditions, all subqueries in NatSQL only appear in its *WHERE* conditions. NatSQL removes the brackets from SQL, so the structure of generated SQL is determined according to the order of conditions of NatSQL and NatSQL only supports the one or two nested subqueries (subquery and sub-subquery), which can cover most of the existing text-to-SQL dataset [Finegan-Dollak et al., 2018].

***SELECT* Clause in Subquery**

Since the subqueries in NatSQL only need one *SELECT* column, we make it meet the *WHERE* condition format. Here is an example:

NatSQL: **SELECT** station.long **WHERE** station.id NOT IN max(status.id)
SQL: **SELECT** long **FROM** station **WHERE** id NOT IN (**SELECT** max(id) **FROM** status)

Since station.id column in the NatSQL can be inferred from the primary and foreign key relationship between table station and status, the station.id column can be removed and replaced with a placeholder @:

NatSQL: **SELECT** station.long **WHERE** @ NOT IN max(status.id)

***ORDER BY* Clause in Subquery**

To the best of our knowledge, *ORDER BY* without *LIMIT* is useless in the subquery of most SQL, because the order of return data from subquery does not affect the result of conditional judgment in the query. However *ORDER BY* with *LIMIT* is useful in a subquery. NatSQL only supports the *ORDER BY* plus *LIMIT* 1 for its subquery, which can cover most subqueries with *ORDER BY* in the Spider dataset. To avoid conflict with the *ORDER BY* in non-subquery, the *ORDER BY* plus *LIMIT* 1 in the subquery is written as a conditional form. For example, "**ORDER BY** age **DESC LIMIT** 1" is written as "age = max(age)" for a subquery. If *ASC* replaces the *DESC*, the "min" will replace the "max" function. Notice that this condition is similar to the example in the last section whose condition is converted into a subquery. To be converted into *ORDER BY*, the following rules should be followed:

- The "W_Oper" (where operator) must be '='.
- The two columns in the condition right and left must be the same.

- The "Cond_R" must contain an aggregate function which is "max" or "min".
- The "connector" before the condition must be "and" or "or".
- This condition must be the end of a subquery.

We define a *ORDER BY* as a conditional form whose function is the same as the representation of a subquery in NatSQL. For example, "*ORDER BY* with *LIMIT* 1 " can be rewritten to a subquery "*WHERE* a = (*SELECT* max(a) ...)". For better understanding, here are some NatSQL examples and their corresponding SQL to illustrate the NatSQL conversion rules.

1.NatSQL:	... **WHERE** b.age = max(c.age) and c.sale = max(c.sale)
1.SQL:	... **WHERE** age = (**SELECT** max(age) **FROM** c **ORDER BY** sale **DESC LIMIT 1**)
2.NatSQL:	... **WHERE** b.age = max(c.age) and c.sale in max(c.sale)
2.SQL:	... **WHERE** age = (**SELECT** max(age) **FROM** c) and sale in (**SELECT** max(sale) **FROM** c)
3.NatSQL:	... **WHERE** b.age = max(c.age) and c.sale = max(d.sale)
3.SQL:	... **WHERE** age = (**SELECT** max(age) **FROM** c) and sale = (**SELECT** max(sale) **FROM** d)
4.NatSQL:	... **WHERE** b.age = max(c.age) and c.sale = max(c.sale) and c.age > 10
4.SQL:	... **WHERE** age = (**SELECT** max(age) **FROM** c) and sale = (**SELECT** max(sale) **FROM** c **WHERE** c.age > 10)

Begin and End of a Subquery

A subquery starts from the "Cond_R" in a *WHERE* condition when "Cond_R" is a "Col_Literal" and when it will not be translated into *ORDER BY* and *JOIN ON* clause as mentioned above. A subquery ends before a new subquery or ends with the NatSQL. A sub-subquery starts after the keyword "sub", and ends before a new subquery or a new sub-subquery, or end with the NatSQL. Figure 4.5 shows two examples.

The rule is that when a subquery begins, all the subsequent conditions belong to it until the end of a subquery. So the NatSQL in Figure 4.5 can be translated into the following queries.

.. **WHERE** age = (**SELECT** max(age) **FROM** t **WHERE** sale > 10) or age = (**SELECT** min(age) **FROM** t **WHERE** sale < 10)

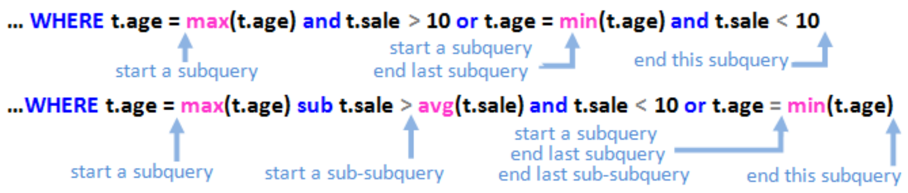


Figure 4.5: Two Examples of The Begin and End of a Subquery

.. **WHERE** age = (**SELECT** max(age) **FROM** t **WHERE** sale > (**SELECT** avg(sale) **FROM** t **WHERE** sale < 10)) or age = (**SELECT** min(age) **FROM** t)

Order of *WHERE* Condition in NatSQL

To simplify SQL, NatSQL removes the brackets which define the structure of a query. To achieve the same structure, NatSQL infers brackets by new keyword "sub" and by identifying the begin and end of a subquery. However, the conditions of subquery need to be placed behind conditions of non-subquery. Fortunately, almost all conditions of subqueries are placed at the end of a query in the original Spider dataset, we think it may due to that subqueries are usually generated by more complex questions that need to be described by longer sentences with clauses. English is used to putting clauses that are usually converted into a subquery on the end of a sentence. Table 4.5 is an example of how the order affects the results.

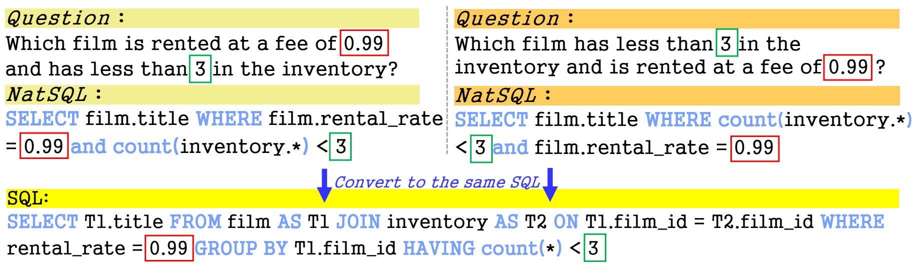
We have considered adding a new condition connector to mark the end of a subquery. If we add it, this issue disappears. However, since the cases affected by this problem are just 6/7000 and 2/1034 in the training and development set of the Spider (there are only 0.1% examples with *WHERE* condition before a subquery in the Spider dataset such as NatSQL 2 in table 4.5). So we decided to ignore this problem now and reorder the *WHERE* condition to avoid it.

Keyword Sub

The keyword "*sub*" is designed for sub-subqueries and a sub-subquery must start after "*sub*". "*sub*" is a condition connector that only connects a sub-subquery whose form is the same as subqueries. Apart from having to start after "*sub*", other rules of sub-subqueries are consistent with subqueries. Here is an example.

NatSQL:	**SELECT** stadium.name **WHERE** stadium.name not in stadium.name and concert.year = 2014
SQL:	**SELECT** name **FROM** stadium **WHERE** stadium.name not in (**SELECT** stadium.name **FROM** concert **JOIN** stadium **ON** concert.stadium_id = stadium.stadium_id **WHERE** concert.year = 2014)
NatSQL:	**SELECT** stadium.name **WHERE** concert.year = 2014 and stadium.name not in stadium.name
SQL:	**SELECT** stadium.name **FROM** concert **JOIN** stadium **ON** concert.stadium_id = stadium.stadium_id **WHERE** concert.year = 2014 and stadium.name not in (**SELECT** name **FROM** stadium)

Table 4.5: How different order of *WHERE* conditions affects NatSQL



```
Question :
Which film is rented at a fee of 0.99
and has less than 3 in the inventory?
NatSQL :
SELECT film.title WHERE film.rental_rate
= 0.99 and count(inventory.*) < 3
```

```
Question :
Which film has less than 3 in the
inventory and is rented at a fee of 0.99 ?
NatSQL :
SELECT film.title WHERE count(inventory.*)
< 3 and film.rental_rate = 0.99
```

```
SQL:
SELECT T1.title FROM film AS T1 JOIN inventory AS T2 ON T1.film_id = T2.film_id WHERE
rental_rate = 0.99 GROUP BY T1.film_id HAVING count(*) < 3
```

Figure 4.6: Fill the values in order of appearance (see more discussion in Appendix B).

NatSQL:	... **WHERE** b.age = max(c.age) sub c.sale = max(c.sale)
SQL:	... **WHERE** age = (**SELECT** max(age) **FROM** c **WHERE** sale = (**SELECT** max(sale) **FROM** c))

4.3.5 Executable SQL Generation

Many previous text-to-SQL models [Guo et al., 2019, Wang et al., 2020, Bogin et al., 2019a] only focus on the Spider exact match accuracy, i.e., they only generate the SQL queries without condition values. These queries are not executable until filling in the condition values. However, it is not easy to fill in the values correctly. On the one hand, there are too many possible condition value slots that need to be searched. The slots can appear in: *WHERE* clause, *WHERE* clause in a subquery, *WHERE* clause after set operators, *HAVING* clause, etc. On the other hand, when there are multiple value slots, it is easier to confuse where to fill. For example, in Figure 4.6, the two different questions correspond to the same SQL query, making it hard to copy the right values from the question to SQL.

Because the condition value slots of NatSQL only appear in the *WHERE* clause, generating condition values becomes much easier, as shown in Figure 4.6. Unlike the models [Lin et al., 2020, Rubin and Berant, 2021] trained to copy the values from questions to SQL queries, NatSQL simply copies the possible values (numbers or database cell values) from questions to SQL in the order of appearance without training. This feature enables the models designed only for the Spider exact match metrics to generate executable SQL.

Training data is the key to ensuring that different questions in Figure 4.6 will be converted to different NatSQL. Firstly, in the dataset, for SQL with multiple *WHERE* conditions, the order of the conditions is mostly consistent with the question. Secondly, the NatSQL further expands this type of training data. For example, the NatSQL queries in Figure 1.4, 4.3, 4.4 contain more *WHERE* conditions than SQL and other IRs, and these conditions appear in the order they are mentioned. These training data make it possible for models to generate different NatSQL according to the different questions in Figure 4.6.

4.4 Limitations of NatSQL

NatSQL removes multiple clauses from SQL that are actually used for special purposes, which means some SQL inevitably cannot be represented by NatSQL. Since there are many equivalent SQL queries, for many unsupported SQL examples, NatSQL can achieve the same functionality by converting to their equivalent format. However, equivalent SQL queries cannot get a positive result in exact set match metrics, introduced in Chapter 2.1.3. The previous sections have partially discussed the limitation of NatSQL due to the deletion of a specific clause. We summarize these limitations here.

JOIN ON. Chapter 4.3.3 discusses three types of JOIN ON clauses incompatible with NatSQL: (1) self-connection; (2) JOIN ON condition without = operator; (3) FROM or JOIN a subquery. The first type can be solved by extending the WHERE condition, similar to the 'join' operator introduced in Chapter 4.2.4. NatSQL cannot generate SQL of the second type, but it can generate SQL with the same functionality by moving the JOIN ON condition to the WHERE clause. In a text-to-SQL process, we are happy with the NatSQL converted to an equivalent SQL that is not the same as the target one. For the third case, there are too few relevant samples for evaluation. As the samples in the Spider, NatSQL can be compatible.

GROUP BY. In the Sqlserver database, the schema item in the GROUP BY clause must appear in the SELECT clause. Therefore, we can generate the

GROUP BY clause based on analyzing the SELECT clause, as discussed in Chapter 4.3.2. However, other databases have no restrictions on the GROUP BY, which may result in a failure to generate the GROUP BY clause. The main error in the current NatSQL to SQL conversion also comes from not being able to generate the correct GROUP BY. We recommend using the NatSQL$_G$ toward a better coverage.

IUE Set Operators. Chapter 4.2.5 discusses how to generate the IUE set operators from NatSQL. Current NatSQL only supports one set operator in the SQL query. Besides, NatSQL assumes that the SELECT clauses in the SQL queries connected by IUE set operators are the same or primary and foreign key relationships. In addition to the above two cases, NatSQL is basically compatible with other SQL with IUE, through direct or equivalent format, unless extremely complex SQL. Since there are few SQL with IUE in the current datasets and no extremely complex SQL, we may miss some compatibility issues.

Subqueries. Chapter 4.3.4 discusses how to generate the subqueries from NatSQL. Currently known NatSQL cannot support subqueries below sub-subqueries, where the sub-subquery is supported by NatSQL keyword sub. As discussed in Chapter 4.3.4, it is useless to sort the subquery results, so NatSQL can only generate its equivalent format. If the subquery is not at the end of an SQL, NatSQL needs to change the subquery position, which is also an equivalent format. Finally, when the subquery needs to attain the maximum and minimum values, and the main query needs to sort the final results, NatSQL can only achieve the same functionality in an equivalent format. Similar to the IUE set operators, we may miss some compatibility issues due to the few SQL with Subqueries in the current datasets.

4.5 Experiments

4.5.1 Experimental Setup

We evaluate NatSQL on the Spider benchmark [Yu et al., 2018b]. There are 7000, 1034 and 2147 samples for training, development and testing respectively, where 206 databases are split into 146 for training, 20 for development and 40 for testing.

We first evaluate the gold NatSQL and other IRs using the exact (set) match and execution match metrics in [Yu et al., 2018b]. Exact (set) match measures whether the predicted query without condition values as a whole is equivalent to the gold query. Execution match measures whether the execution result of the

Language	Exact Match	Execution Match
SQL	100%	100%
SemQL	86.2%	Unsupported
IR(RAT-SQL)	97.7%	97.1%
NatSQL	93.3%	95.3%
NatSQL$_{\mathbf{G}}$	96.2%	96.5%

Table 4.6: The comparison between gold IRs on Spider development set.

Ques:	Find students whose age is 10 or 16.
SQL 1:	... **WHERE** age = 10 **or** age = 16
NatSQL 1:	... **WHERE** age = 10 **or** age = 16
SQL 2:	... **WHERE** age = 10 **UNION** ... **WHERE** age = 16
NatSQL 2:	... **WHERE** age = 10 **union** age = 16

Table 4.7: Equivalent SQL queries with its NatSQL

predicted query from the database is the same as the gold query. For the details of exact (set) match and execution match, please refer to Chapter 2.1.3. We then evaluate NatSQL and other IRs using existing open-source models that provide competitive performance on Spider: (1) GNN [Bogin et al., 2019a]; (2) IRNet [Guo et al., 2019]; (3) RAT-SQL [Wang et al., 2020]; (4) RAT-SQL+GAP [Shi et al., 2021]. Although some of these models are not designed for the generation of executable SQL queries, with the approach discussed in Section 4.3.5, we utilize NatSQL to generate executable SQL and evaluate the execution match performance.

4.5.2 Comparison Between IRs

Gold IRs

In Table 4.6, we present the exact match and execution match accuracies of the gold IRs on the Spider development set, where the metrics are defined by Yu et al. [2018b] for the Spider benchmark.

We observe that NatSQL can be converted to more gold SQL than SemQL, because NatSQL can handle the unremovable *JOIN ON* clauses, as discussed in Section 4.2.4. Such SQL queries comprise around 5% of the entire Spider dataset. Other performance improvement comes from the fact that NatSQL is more compatible with subqueries and that its capability to generate SQL is better. More importantly, SemQL is designed only for the exact match metrics of Spider, and cannot directly be used to generate executable SQL.

Component	F1	Component	F1
select	0.997	where	0.969
group	0.879	order	0.996
and/or	0.998	IUE	0.900
keywords	0.989		

Table 4.8: Partial matching F1 score of NatSQL on the Spider development set.

The IR of RAT-SQL is the most similar to SQL and thus has the highest coverage among all IRs. However, NatSQL$_{\mathbf{G}}$ further simplifies the queries with only 0.6% execution accuracy degradation, whilst enabling better model prediction performance. NatSQL$_{\mathbf{G}}$ outperforms NatSQL when comparing the gold queries, but the gap is small when they are utilized by models.

The result in the training set is close to that in the development set. It should be noted that the exact match accuracy will slightly vary in different NatSQL versions. The accuracy depends on the attitude towards equivalent SQL queries. Table 4.7 presents two equivalent SQL queries with their corresponding NatSQL queries. Considering that *UNION* is not mentioned in the question, we prefer to sacrifice the exact match accuracy for a more succinct NatSQL representation, i.e., we will use the first NatSQL query in Table 4.7 to represent the second SQL, even though it can not be converted into the second SQL query. Although our preference slightly affects the exact match accuracy in the Spider benchmark, it brings greater potential and convenience when outside Spider.

Gold NatSQL Error Analysis

Table 4.8 presents the F1 score of NatSQL for different SQL components. We observe that the main errors come from *GROUP BY* and IUE matching. Although NatSQL cannot be converted to all gold *GROUP BY* clauses, most of these errors don't affect the execution results. The IUE errors occur because NatSQL only supports one IUE operator per query.

Some other errors are due to the limitation of the exact match evaluation method when evaluating the *JOIN ON* clause of subqueries and sub-subqueries. Specifically, when the *FROM* and *JOIN* in a generated subquery is not identical to the gold SQL, the Spider evaluation scheme considers it to be wrong. For example, the following two SQL statements have the same semantic meaning, but they are recognized as different by the Spider exact match evaluation method, thus results in an exact match error.

... col in (**SELECT** col **FROM** T1 **JOIN** T2 ...)

... col in (**SELECT** col **FROM** T2 **JOIN** T1 ...)

NatSQL Coverage for Different Text-to-SQL Datasets

Along with the Spider, Yu et al. [2018b] also provides NL question and SQL pairs extracted from other text-to-SQL datasets, which can be used to evaluate the coverage of NatSQL. However, the database in some datasets has limited data, which leads to the different SQL queries getting positive results by obtaining the same none data return. Therefore, it is not suitable to use the execution match here. Table 4.9 presents the exact match accuracy of gold NatSQL in four text-to-SQL datasets. In particular, since WikiSQL only contains simple SQL queries, NatSQL can be compatible with the entire dataset. Different from WikiSQL, examples of NatSQL incompatibility appear in other datasets. In GeoQuery, the main problem in NatSQL is unsupported the too complex subqueries. In IMDB, MAS, and MAS, the self join connection causes major incompatibility issues.

Dataset	Exact Match
GeoQuery	90.0%
IMDB	93.6%
YELP	82.9%
MAS	90.4%
WikiSQL	100%

Table 4.9: NatSQL coverage for different text-to-SQL datasets that are introduced in Chapter 3.4.1

IRs for Prediction

Table 4.10 presents the exact match accuracy of four models with SemQL, its default IR (or SQL), and NatSQL separately. We observe that NatSQL consistently outperforms SemQL with all of these model architectures, including IRNet. Note that the original Spider dataset additionally includes 1,659 training samples from 6 earlier text-to-SQL benchmarks (Academic, GeoQuery, IMDB, Restaurants, Scholar and Yelp), which were used to train models with SemQL in the IRNet. To provide a fair comparison with other baselines, we didn't include these additional samples for all models in our evaluation, thus our presented result for IRNet+SemQL (51.8%) is lower than the number reported in the IRNet paper (53.2%).

Note that SemQL causes performance decline for RAT-SQL. We hypothesize that this is because the exact match accuracy of the gold SemQL is only 86.2%. With the improvement of model architectures, such a gap will affect the prediction accuracy more negatively. Although the accuracy of gold RAT-SQL IR is higher than that of NatSQL, NatSQL still outperforms the original RAT-SQL

Approach	Exact	Execution
GNN + SQL	47.5%	
GNN + SemQL	51.6%	
GNN + NatSQL	**53.8%**	**58.0%**
IRNet + SemQL	51.8%	
IRNet + NatSQL	**52.9%**	**52.6%**
RAT-SQL + IR(RAT-SQL)	62.7%	
RAT-SQL + SemQL	58.4%	
RAT-SQL + NatSQL	64.4%	66.7%
RAT-SQL + NatSQL$_\mathbf{G}$	**65.2%**	**67.3%**
extend BERT:		
RAT-SQL + IR(RAT-SQL)	69.5%	
RAT-SQL + NatSQL	71.7%	72.8%
RAT-SQL + NatSQL$_\mathbf{G}$	**72.1%**	**73.0%**
extend GAP:		
RAT-SQL + IR(RAT-SQL)	71.8%	
RAT-SQL + NatSQL	**73.7%**	74.6%
RAT-SQL + NatSQL$_\mathbf{G}$	**73.7%**	**75.0%**

Table 4.10: Exact and execution match accuracy on Spider development set.

model, and NatSQL$_\mathbf{G}$ slightly improves the performance over NatSQL.

Meanwhile, NatSQL helps these models generate executable SQL queries. Execution match accuracy improves with the improvement of the exact match, and most execution match accuracy is better than that of exact match. The execution match accuracy of IRNet is slightly lower than the exact match, because the IRNet does not predict the *DISTINCT* keyword while the exact match metric does not check this aspect.

Breakdown results. Based on the complexity of the SQL, the examples in Spider are classified into four types: `easy`, `medium`, `hard`, and `extra hard`. We provide a breakdown comparison on the Spider development set, as shown in Table 4.11. The improvement brought by NatSQL mainly comes from the `extra hard` SQL, which demonstrate an average 4.74% absolute improvement across these models. This improvement is in line with the design of NatSQL, i.e., most `extra hard` SQL queries contain set operators or subqueries, while NatSQL has simplified these components. Since `easy` and `medium` SQL queries categorized in the Spider dataset are more similar to NatSQL queries, it is expected that the improvement on simple SQL is less significant. However, we still observe that NatSQL consistently increases the accuracy on most samples of different difficulty levels.

Approach	Easy	Medium	Hard	Extra
GNN + SemQL	68.5%	**58.9%**	36.8%	24.1%
GNN + NatSQL	**72.0%**	58.0%	**42.0%**	**28.2%**
IRNet + SemQL	69.8%	53.0%	**46.0%**	30.1%
IRNet + NatSQL	**70.6%**	**54.1%**	**46.0%**	**32.5%**
RAT-SQL + IR(RAT-SQL)	80.4%	63.9%	55.7%	40.6%
RAT-SQL + $\text{NatSQL}_{\mathbf{G}}$	**82.4%**	**65.0%**	**59.2%**	**46.5%**
extend BERT:				
RAT-SQL + IR(RAT-SQL)	86.4%	73.6%	62.1%	42.9%
RAT-SQL + $\text{NatSQL}_{\mathbf{G}}$	**88.4%**	**76.6%**	**62.6%**	**46.4%**
extend GAP:				
RAT-SQL + IR(RAT-SQL)	88.3%	74.0%	64.4%	44.0%
RAT-SQL + $\text{NatSQL}_{\mathbf{G}}$	**91.6%**	**75.2%**	**65.5%**	**51.8%**

Table 4.11: Exact match accuracy by difficulty on Spider development set.

4.5.3 Overall Performance Analysis

First, we present the exact and execution match accuracy of our approach applied to RAT-SQL augmented with GAP in Table 4.12, where we compare with various baselines at the top of the Spider leaderboard. By incorporating NatSQL into the RAT-SQL model with GAP, we demonstrate that our approach achieves a new state-of-the-art on Spider execution benchmark, surpassing its best counterparts by 2.2% absolute improvement.

Considering that the gap between dev and test in exact match is larger than that in execution match, we speculate that there are two reasons why our exact match accuracy has dropped by 1% compared to RAT-SQL+GAP. From the complexity breakdown accuracy between dev and test, we observe that the main performance degradation comes from the `extra hard` SQL queries. Since there are many subqueries in `extra hard` SQL queries, some limitations of the Spider exact match evaluation process (discussed in Chapter 4.5.2) may have a negative effect on our prediction results. On the other hand, some degradation may come from equivalent SQL queries. As we discuss in Section 4.5.2 and Table 4.7, it is not mandatory to keep the NatSQL queries consistent with the original SQL queries. As a result, the model trained by NatSQL may output equivalent SQL queries that do not match exactly but that get the same query result. Therefore, our evaluation shows that NatSQL is more suitable for generating executable SQL queries.

4.6 Summary

In this chapter, we propose NatSQL, a new SQL intermediate representation that reduces the difficulty of schema linking and simplifies the SQL structure.

Approach	**Exact**	**Execution**
IRNet + BERT [Guo et al., 2019]	54.7%	–
RATSQL + BERT [Wang et al., 2020]	65.6%	–
BRIDGE v2 + BERT(ensemble) [Lin et al., 2020]	67.5%	68.3%
COMBINE (Anonymous)	67.7%	68.2%
SmBoP + GraPPa [Rubin and Berant, 2021]	69.5%	71.1%
RATSQL + GAP [Shi et al., 2021]	69.7%	–
DT-Fixup SQL-SP + RoBERTa (Anonymous)	**70.9%**	–
RAT-SQL + GAP + NatSQL$_{\mathbf{G}}$ (Ours)	68.7%	**73.3%**

Table 4.12: Results on Spider test set, compared to other models at the top of the leaderboard.

By incorporating NatSQL into existing neural models for text-to-SQL generation, we show that NatSQL is easier to infer from natural language specification than the full-fledged SQL and other intermediate representation languages. Furthermore, NatSQL enables existing models to easily generate executable SQL queries without modifying their architecture. Experimental results on the challenging Spider benchmark demonstrate that NatSQL consistently improves the prediction performance of several neural network architectures and achieves the state-of-the-art, showing the effectiveness of our approach.

Chapter 5

Text-to-SQL Robustness against Synonym Substitution

In Chapter 4, we introduce the NatSQL and improve the previous state-of-the-art models achieving around 70% accuracy on the Spider test set, even if the model is tested on databases unseen during training. However, we suspect that such cross-domain generalization heavily relies on the exact lexical matching between the NL question and the table schema. As shown in Figure 1.5, names of tables and columns in the SQL query are explicitly stated in the NL question. Such questions constitute the majority of cross-domain text-to-SQL benchmarks, including both Spider and WikiSQL. Although assuming exact lexical matching is a good starting point to solving the text-to-SQL problem, this assumption usually does not hold in real-world scenarios. Specifically, it requires that users have precise knowledge of the table schemas to be included in the SQL query, which could be tedious for synthesizing complex SQL queries.

In this chapter, we investigate whether state-of-the-art text-to-SQL models preserve good prediction performance without the assumption of exact lexical matching, where NL questions use synonyms to refer to tables or columns in SQL queries. We call such NL questions *synonym substitution* questions. Although some existing approaches can automatically generate synonymous substitution examples, these examples may deviate from real-world scenarios, e.g., they may not follow common human writing styles, or even accidentally becomes inconsistent with the annotated SQL query. To provide a reliable benchmark for evaluating model performance on synonym substitution questions, we introduce Spider-Syn, a human-curated dataset constructed by modifying NL questions in the Spider dataset. Specifically, we replace the schema annotations in the NL question with synonyms, manually selected so as not to change the corresponding SQL query, as shown in Figure 1.5. We demonstrate that when models are only trained on the original Spider dataset, they suffer a significant performance drop on Spider-Syn, even though the Spider-Syn benchmark is not constructed to exploit the worst-case attacks for text-to-SQL models. It is therefore clear that the performance of these models will suffer in real-world use, particularly in cross-domain scenarios.

To improve the robustness of text-to-SQL models, we utilize synonyms of table schema words, which are either manually annotated, or automatically generated when no annotation is available. We investigate two categories of approaches to incorporate these synonyms. The first category of approaches modify the schema annotations of the model input, so that they align better with the NL question. No additional training is required for these approaches. The second category of approaches are based on adversarial training, where we augment the training set with NL questions modified by synonym substitution. Both categories of approaches significantly improve the robustness, and the first category is both effective and requires less computational resources.

This chapter is based on [Gan et al., 2021a]. Our contributions in this chapter are as follows:

- We conduct a comprehensive study to evaluate the robustness of text-to-SQL models against synonym substitution.
- Besides worst-case adversarial attacks, we further introduce Spider-Syn, a human-curated dataset built upon Spider, to evaluate synonym substitution for real-world question paraphrases.
- We propose a simple yet effective approach to utilize multiple schema annotations, without the need of additional training. We show that our approach outperforms adversarial training methods on Spider-Syn, and achieves competitive performance on worst-case adversarial attacks.

5.1 Spider-Syn Dataset

5.1.1 Overview

We construct the Spider-Syn benchmark by manually modifying NL questions in the Spider dataset using synonym substitution. The purpose of building Spider-Syn is to simulate the scenario where users do not call the exact schema words in the utterances, e.g., users may not have the knowledge of table schemas. In particular, we focus on synonym substitution for words related to databases,

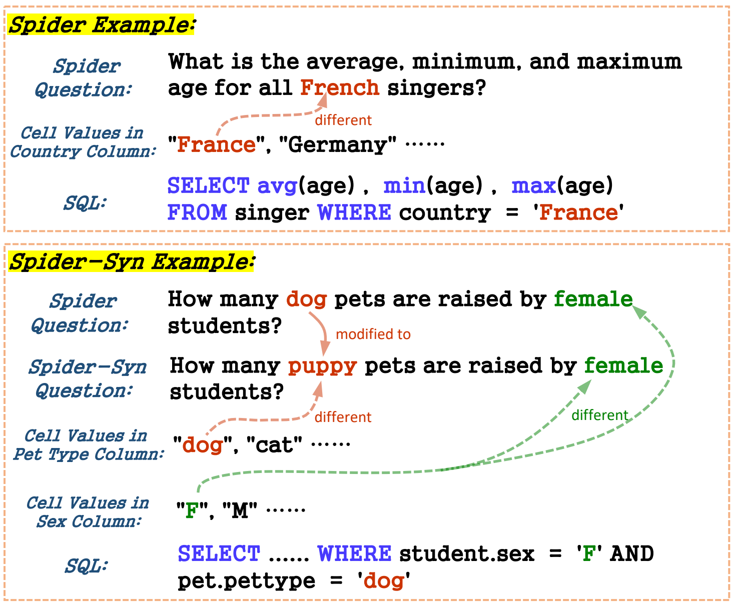

Spider Example:
Spider Question:
What is the average, minimum, and maximum age for all French singers?
different
Cell Values in Country Column:
"France", "Germany" ......
SQL:
SELECT avg(age), min(age), max(age) FROM singer WHERE country = 'France'
Spider−Syn Example:
Spider Question:
How many dog pets are raised by female students?
modified to
Spider−Syn Question:
How many puppy pets are raised by female students?
different
different
Cell Values in Pet Type Column:
"dog", "cat" ......
Cell Values in Sex Column:
"F", "M" ......
SQL:
SELECT ...... WHERE student.sex = 'F' AND pet.pettype = 'dog'

Figure 5.1: Synonym substitution occurs in cell value words in both Spider and Spider-Syn.

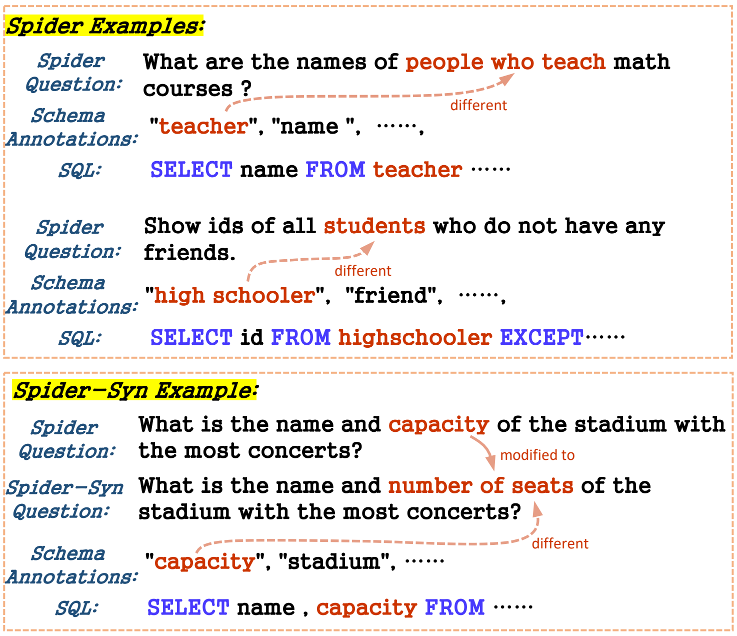
including table schemas and cell values. Consistent with Spider, Spider-Syn contains 7000 training and 1034 development examples, but Spider-Syn does not contain a test set since the Spider test set is not public. Figure 1.5 presents two examples in Spider-Syn and how they are modified from Spider.

5.1.2 Conduct Principle

The goal of constructing the Spider-Syn dataset is not to perform worst-case adversarial attacks against existing text-to-SQL models, but to investigate the model robustness for paraphrasing schema-related words, which is particularly important when users do not have the knowledge of table schemas. We carefully select the synonyms to replace the original text to ensure that new words will not cause ambiguity in some domains. For example, the word '*country*' can often be used to replace the word '*nationality*'. However, we did not replace it in the domain whose '*country*' means people's '*born country*' different from its other schema item, '*nationality*'. Besides, some synonym substitutions are only valid in the specific domain. For example, the word '*number*' and '*code*' are not generally synonymous, but '*flight number*' can be replaced by '*flight code*' in the aviation domain.

Most synonym substitutions use relatively common words[1] to replace the

[1]According to 20,000 most common English words in `https://github.com/first20hours/google-10000-english`.



Spider Examples:

Spider Question: What are the names of people who teach math courses ?

Schema Annotations: "teacher", "name ", ……,

SQL: SELECT name FROM teacher ……

Spider Question: Show ids of all students who do not have any friends.

Schema Annotations: "high schooler", "friend", ……,

SQL: SELECT id FROM highschooler EXCEPT……

Spider−Syn Example:

Spider Question: What is the name and capacity of the stadium with the most concerts?

Spider−Syn Question: What is the name and number of seats of the stadium with the most concerts?

Schema Annotations: "capacity", "stadium", ……

SQL: SELECT name , capacity FROM ……

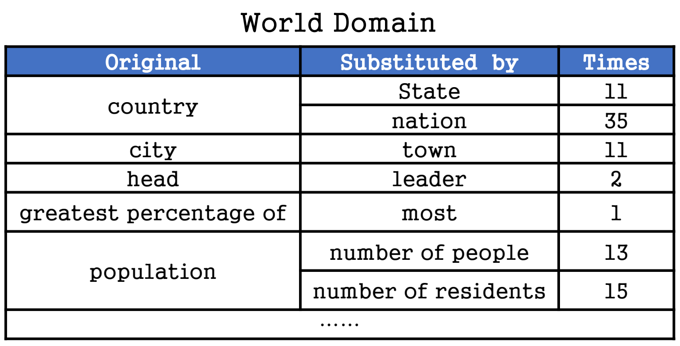
Figure 5.2: Samples of replacing the original words or phrases by synonymous phrases.

schema item words. Besides, we denote '*id*', '*age*', '*name*', and '*year*' as reserved words, which are the most standard words to represent their meanings. Under this principle, we keep some original Spider examples unchanged in Spider-Syn. Our synonym substitution does not guarantee that the modified NL question has the exact same meaning as the original question, but guarantees that its corresponding SQL is consistent. In Figure 5.1, Spider-Syn replaces the cell value word '*dog*' with '*puppy*'. Although puppy is only a subset of dog, the corresponding SQL for the Spider-Syn question should still use the word '*dog*' instead of the word '*puppy*' because there is only dog type in the database and no puppy type. Similar reasoning is needed to infer that the word '*female*' corresponds to '*F*' in Figure 5.1.

In some cases, words are replaced by synonymous phrases (rather than single words), as shown in Figure 5.2. Besides, some substitutions are also based on the database contents. For example, a column '*location*' of the database '*employee_hire_evaluation*' in Spider only stores city names as cell values. Without knowing the table schema, users are more likely to call '*city*' instead of '*location*' in their NL questions.

To summarize, we construct Spider-Syn with the following principles:

- Spider-Syn is not constructed to exploit the worst-case adversarial attacks, but to represent real-world use scenarios; it therefore uses only relatively



World Domain

Original	Substituted by	Times
country	State	11
	nation	35
city	town	11
head	leader	2
greatest percentage of	most	1
population	number of people	13
	number of residents	15
......		

Figure 5.3: Examples of synonym substitutions in the 'world' domain from Spider-Syn.

common words as substitutions.

- We conduct synonym substitution only for words related to schema items and cell values.
- Synonym substitution includes both single words and phrases with multiple words.

5.1.3 Annotation Steps

Before annotation, we first separate original Spider samples based on their domains. For each domain, we only utilize synonyms that are suitable for that domain. We recruit four graduate students major in computer science to annotate the dataset manually. They are trained with a detailed annotation guideline, principles, and some samples. One is allowed to start after his trial samples are approved by the whole team.

As synonyms can be freely chosen by annotators, standard inter-annotator agreement metrics are not sufficient to confirm the data quality. Instead, we conduct the quality control with two rounds of review. The first round is the cross-review between annotations. We require the annotators to discuss their disagreed annotations and come up with a final result out of consensus. To improve the work efficiency, we extract all synonym substitutions as a report without the NL questions from the annotated data, as shown in Figure 5.3. Then, the annotators do not have to go through the NL questions one by one. The second round of review is similar to the first round but is done by native English speakers.

5.1.4 Dataset Statistics

In Spider-Syn, 5672 questions are modified compared to the original Spider dataset. In 5634 cases the schema item words are modified, with the cell value words modified in only 27 cases.We use 273 synonymous words and 189 synonymous phrases to replace approximately 492 different words or phrases in these questions. In all Spider-Syn examples, there is an average of 0.997 change per question and 7.7 words or phrases modified per domain.

Besides, Spider-Syn keeps 2201 and 161 original Spider questions in the training and development set, respectively. In the modification between the training and development sets, 52 modified words or phrases were the same, accounting for 35% of the modification in the development set.

5.2 Defense Approaches

We present two categories of approaches for improving model robustness to synonym substitution. We first introduce our multiple annotation selection approach, which could utilize multiple annotations for one schema item. Then we present an adversarial training method based on analysis of the NL question and domain information.

5.2.1 Multi-Annotation Selection (MAS)

The synonym substitution problem emerges when users do not call the exact names in table schemas to query the database. Therefore, one defense against synonym substitution is utilizing multiple annotation words to represent the table schema, so that the schema linking mechanism is still effective. For example, for a database table with the name '*country*', we annotate additional table names with similar meanings, e.g., '*nation*', '*State*', etc. In this way, we explicitly inform the text-to-SQL models that all these words refer to the same table, thus the table should be called in the SQL query when the NL question includes any of the annotated words.

We design a simple yet effective mechanism to incorporate multiple annotation words, called multiple-annotation selection (MAS). For each schema item, we check whether any annotations appear in the NL question, and we select such annotations as the model input. When no annotation appears in the question, we select the default schema annotation, i.e., the same as the original Spider dataset. In this way, we could utilize multiple schema annotations simultaneously, without changing the model input format.

The main advantage of this method is that it does not require additional training, and could apply to existing models trained without synonym substitu-

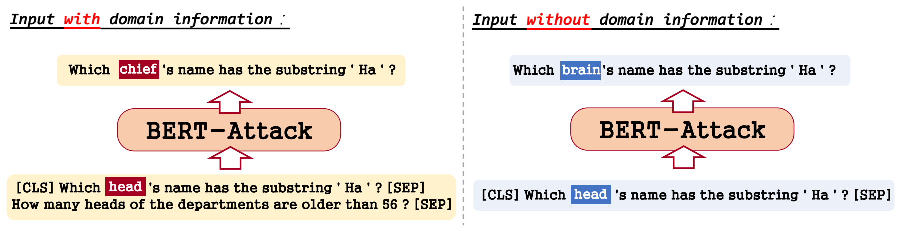

Input with domain information :
Which chief 's name has the substring ' Ha ' ?
BERT−Attack
[CLS] Which head 's name has the substring ' Ha ' ? [SEP]
How many heads of the departments are older than 56 ? [SEP]
Input without domain information :
Which brain 's name has the substring ' Ha ' ?
BERT−Attack
[CLS] Which head 's name has the substring ' Ha ' ? [SEP]

Figure 5.4: Input the BERT-Attack with and without domain information.

tion questions. Annotating multiple schema words could be done automatically or manually, and we compare them in Section 5.3.

5.2.2 Adversarial Training

Motivated by the idea of adversarial training that can improve the robustness of machine learning models against adversarial attacks [Madry et al., 2018, Morris et al., 2020], we implement adversarial training using the current open-source SOTA model RAT-SQL [Wang et al., 2020]. We use the BERT-Attack model [Li et al., 2020] to generate adversarial examples, and implement the entire training process based on the TextAttack framework [Morris et al., 2020]. TextAttack provides 82 pre-trained models, including word-level LSTM, word-level CNN, BERT-Attack, and other pre-trained Transformer-based models.

We follow the standard adversarial training pipeline that iteratively generates adversarial examples, and trains the model on the dataset augmented with these adversarial examples. When generating adversarial examples for training, we aim to generate samples that align with the Spider-Syn principles, rather than arbitrary adversarial perturbations. We describe the details of adversarial example generation below.

Generating Adversarial Examples

We choose BERT-Attack to generate the adversarial examples. Different from other word substitution methods [Mrkšić et al., 2016, Ebrahimi et al., 2018, Wei and Zou, 2019], BERT-Attack model considers the entire NL question when generating words for synonym substitution. Such a sentence-based method can generate different synonyms for the same word in different context. For example, the word '*head*' in '*the head of a department*' and '*the head of a body*' should correspond to different synonyms. Making such distinctions requires an analysis of the entire sentence, since the keywords' positions may not be close, such as that the word '*head*' and '*department*' are not close in '*Give me the info of heads whose name is Mike in each department*'.

In addition to the original question, we add extra domain information into the BERT-Attack model, as shown in Figure 5.4. Without the domain information, on the right side of the Figure 5.4, the BERT-Attack model conjectures the word '*head*' represent the head of a body, since there are multiple feasible interpretations for the word '*head*' if you only look at the question. To eliminate the ambiguity, we feed questions with its domain information into the BERT-Attack model, as shown on the left side of the Figure 5.4.

Instead of using schema annotations, we select several other questions from the same domain as domain information. These questions should contain the schema item words we plan to replace, and other distinct schema item words in the same domain. The benefits of using sentences instead of schema annotations as domain information include: 1) avoiding many unrelated schema annotations, which could include hundreds of words; 2) the sentence format is closer to the pre-training data of BERT. As shown on the left side of the Figure 5.4, our method improves the quality of data generation.

Since we focuses on the synonym substitution of schema item words, we make two additional constraints to limit the generation of adversarial examples: 1) only words about schema items and cell values can be replaced; and 2) do not replace the reserved words discussed in Section 5.1.2. These constraints make sure that the adversarial examples only perform the synonym substitution for words related to database tables.

5.3 Experiments

5.3.1 Experimental Setup

We compare our approaches against baseline methods on both the Spider [Yu et al., 2018b] and Spider-Syn development sets. As discussed in Section 5.1.1, the Spider test set is not publicly accessible, and thus Spider-Syn does not contain a test set. Both Spider and Spider-Syn contain 7000 training and 1034 development samples respectively, where there are 146 databases for training and 20 for development. The SQL queries and schema annotations between Spider and Spider-Syn are the same; the difference is that the questions in Spider-Syn are modified from Spider by synonym substitution. Models are evaluated using the official exact matching accuracy metric of Spider.

We first evaluate open-source models that reach competitive performance on Spider: GNN [Bogin et al., 2019a], IRNet [Guo et al., 2019] and RAT-SQL [Wang et al., 2020], on the Spider-Syn development set. We then evaluate our approaches with RAT-SQL+BERT model (denoted as RAT-SQL$_{\mathrm{B}}$) on both Spider-Syn and Spider development set.

We examine the robustness of following approaches for synonym substitution:

- **SPR:** Indicate that the model is trained on the Spider dataset.
- **SPR$_{\textbf{SYN}}$:** Indicate that the model is trained on the Spider-Syn dataset .
- **SPR$_{\textbf{SPR\&SYN}}$:** Indicate that the model is trained on both Spider and Spider-Syn datasets.
- **ADV$_{\textbf{BERT}}$:** To improve the robustness of text-to-SQL models, we use adversarial training methods to deal with synonym substitution. This variant means that we use BERT-Attack following the design introduced in Section 5.2.2. Note that we only use the Spider dataset for adversarial training.
- **ADV$_{\textbf{GLOVE}}$:** To demonstrate the effectiveness of our ADV$_{\text{BERT}}$ method, we also evaluate a simpler adversarial training method based on the nearest GLOVE word vector [Pennington et al., 2014, Mrkšić et al., 2016]. This method only considers the meaning of a single word, dispensing with domain information and question context.
- **ManualMAS:** MAS stands for '*multi-annotation selection*', as introduced in Section 5.2.1. ManualMAS means that we collect multiple annotations of schema item words, which are synonyms used in Spider-Syn. Afterward, MAS selects the appropriate annotation for each schema item as the model input.
- **AutoMAS:** In contrast to ManualMAS, in AutoMAS we collect multiple annotations based on the nearest GLOVE word vector, as used in ADV$_{\text{GLOVE}}$. In this way, compared to ManualMAS, there are much more synonyms to be selected from for AutoMAS. Both ManualMAS and AutoMAS are to demonstrate the effectiveness of MAS in an ideal case. This experimental design principle is similar to evaluating adversarially trained models on the same adversarial attack used for training, which aims to show the generalization to in-distribution test samples.

5.3.2 Results of Models Trained on Spider

Table 5.1 presents the exact matching accuracy of models trained on the Spider training set, and we evaluate them on development sets of Spider and Spider-Syn. Although Spider-Syn is not designed to exploit the worst-case attacks of text-to-SQL models, compared to Spider, the performance of all models has clearly dropped by about 20% to 30% on Spider-Syn. Using BERT for input embedding suffers less performance degradation than models without BERT, but

model	Spider	Spider-Syn
GNN + SPR [Bogin et al., 2019a]	48.5%	23.6%
IRNet + SPR [Guo et al., 2019]	53.2%	28.4%
RAT-SQL + SPR [Wang et al., 2020]	62.7%	33.6%
RAT-SQL$_B$ + SPR [Wang et al., 2020]	69.7%	48.2%

Table 5.1: Exact match accuracy on the Spider and Spider-Syn development set, where models are trained on the original Spider training set.

SQL Component	Spider	Spider-Syn
SELECT	0.910	0.699
SELECT (no AGG)	0.926	0.712
WHERE	0.772	0.715
WHERE (no OP)	0.824	0.757
GROUP BY (no HAVING)	0.846	0.575
GROUP BY	0.816	0.553
ORDER BY	0.831	0.768
AND/OR	0.979	0.977
IUE	0.550	0.344
KEYWORDS	0.897	0.876

Table 5.2: F1 scores of component matching of RAT-SQL$_B$+SPR on development sets.

the drop is still significant. These experiments demonstrate that training on Spider alone is insufficient for achieving good performance on synonym substitutions, because the Spider dataset only contains a few questions with synonym substitution.

To obtain a better understanding of prediction results, we compare the F1 scores of RAT-SQL$_B$+SPR on different SQL components on both the Spider and Spider-Syn development set. As shown in Table 5.2, the performance degradation mainly comes from the components including schema items, while the decline in the '*KEYWORDS*' and the '*AND/OR*' that do not include schema items is marginal. This observation is consistent with the design of Spider-Syn, which focuses on the substitution of schema item words.

5.3.3 Comparison of Different Approaches

Table 5.3 presents the results of RAT-SQL$_B$ trained with different approaches. We focus on RAT-SQL$_B$ since it achieves the best performance on both Spider and Spider-Syn, as shown in Table 5.1. Our MAS approaches significantly improve the performance on Spider-Syn, with only 1-2% performance degradation on the Spider. With ManualMAS, we see an accuracy of 62.6%, which

Approach	**Spider**	**Spider-Syn**
SPR	**69.7%**	48.2%
SPR_{SYN}	67.8%	59.9%
$\text{SPR}_{\text{SPR\&SYN}}$	68.1%	58.0%
$\text{ADV}_{\text{GLOVE}}$	48.7%	27.7%
ADV_{BERT}	68.7%	58.5%
SPR + ManualMAS	67.4%	**62.6%**
SPR + AutoMAS	68.7%	56.0%

Table 5.3: Exact match accuracy on the Spider and Spider-Syn development set. All approaches use the $\text{RAT-SQL}_{\text{B}}$ model.

Approach	$\textbf{ADV}_{\textbf{GLOVE}}$	$\textbf{ADV}_{\textbf{BERT}}$
SPR	38.0%	48.8%
SPR_{SYN}	49.6%	54.9%
$\text{SPR}_{\text{SPR\&SYN}}$	47.7%	55.7%
$\text{ADV}_{\text{GLOVE}}$	29.7%	33.8%
ADV_{BERT}	55.7%	**59.2%**
SPR + ManualMAS	34.2%	44.5%
SPR + AutoMAS	**61.2%**	52.5%

Table 5.4: Exact match accuracy on the worst-case development sets generated by $\text{ADV}_{\text{GLOVE}}$ and ADV_{BERT}. All approaches use the $\text{RAT-SQL}_{\text{B}}$ model.

outperforms all other approaches evaluated on Spider-Syn.

We compare the result of $\text{RAT-SQL}_{\text{B}}$ trained on Spider (SPR) as a baseline with other approaches. $\text{RAT-SQL}_{\text{B}}$ trained on Spider-Syn (SPR_{SYN}) obtains 11.7% accuracy improvement when evaluated on Spider-Syn, while only suffers 1.9% accuracy drop when evaluated on Spider. Meanwhile, our adversarial training method based on BERT-Attack (ADV_{BERT}) improves the accuracy by 10.3% on Spider-Syn. We observe that ADV_{BERT} performs much better than adversarial training based on GLOVE ($\text{ADV}_{\text{GLOVE}}$), and we provide more explanation in Section 5.3.4. Both of our multiple annotation methods (ManualMAS and AutoMAS) improve the baseline model evaluated on Spider-Syn. The performance of ManualMAS is better because the synonyms in ManualMAS are exactly the same as the synonym substitution in Spider-Syn. We discuss more results about multi-annotation selection in Section 5.3.5.

5.3.4 Evaluation on Adversarial Attacks

Observing the dramatic performance drop on Spider-Syn, we then study the model robustness under worst-case attacks. We use the adversarial examples

Approach	**Spider**	**Spider-Syn**	**$\text{ADV}_{\text{GLOVE}}$**	**ADV_{BERT}**
SPR	**69.7%**	48.2%	38.0%	48.8%
SPR + ManualMAS	67.4%	**62.6%**	34.2%	44.5%
SPR + AutoMAS	68.7%	56.0%	**61.2%**	**52.5%**
SPR_{SYN}	**67.8%**	59.9%	49.6%	**54.9%**
SPR_{SYN} + ManualMAS	65.7%	**62.9%**	47.8%	52.1%
SPR_{SYN} + AutoMAS	67.0%	61.7%	**63.3%**	54.4%
SPR&SPR_{SYN}	**68.1%**	58.0%	47.7%	**55.7%**
SPR&SPR_{SYN} + ManualMAS	65.6%	**59.5%**	46.9%	51.7%
SPR&SPR_{SYN} + AutoMAS	66.8%	57.5%	**61.0%**	**55.7%**
ADV_{BERT}	**68.7%**	58.5%	55.7%	**59.2%**
ADV_{BERT} + ManualMAS	66.7%	**62.2%**	53.4%	56.7%
ADV_{BERT} + AutoMAS	67.5%	59.6%	**62.4%**	58.0%

Table 5.5: Ablation study results using $\text{RAT-SQL}_{\text{B}}$.

generation module in $\text{ADV}_{\text{GLOVE}}$ and ADV_{BERT} to attack the $\text{RAT-SQL}_{\text{B}}$+SPR to generate two worst-case development sets.

Table 5.4 presents the results on two worst-case development sets. The $\text{ADV}_{\text{GLOVE}}$ and ADV_{BERT} attacks cause the accuracy of $\text{RAT-SQL}_{\text{B}}$+SPR to drop by 31.7% and 20.9%, respectively. $\text{RAT-SQL}_{\text{B}}$+SPR+AutoMAS achieve the best performance on defending the $\text{ADV}_{\text{GLOVE}}$ attack. Because the annotations in AutoMAS cover the synonym substitutions generated by $\text{ADV}_{\text{GLOVE}}$. The relation between AutoMAS and $\text{ADV}_{\text{GLOVE}}$ is similar to that between ManualMAS and Spider-Syn. Similarly, ManualMAS helps $\text{RAT-SQL}_{\text{B}}$+SPR get the best accuracy as shown in Table 5.3.

As to ADV_{BERT} attack, $\text{RAT-SQL}_{\text{B}}$+$\text{ADV}_{\text{BERT}}$ outperforms other approaches. This result is not surprising, because $\text{RAT-SQL}_{\text{B}}$+$\text{ADV}_{\text{BERT}}$ is trained based on defense ADV_{BERT} attack. However, why does $\text{RAT-SQL}_{\text{B}}$ + $\text{ADV}_{\text{GLOVE}}$ perform so poorly in defending $\text{ADV}_{\text{GLOVE}}$ attack?

We conjecture that this is because the word embedding from BERT is based on the context: if you replace a word with a so-called synonym that is irrelevant to the context, BERT may give this synonym a vector with low similarity to the original. In the first example of Table 5.6, $\text{ADV}_{\text{GLOVE}}$ replaces the word '*courses*' with '*trajectory*'. We observe that, based on the cosine similarity of BERT embedding, the schema item most similar to '*trajectory*' changes from '*courses*' to '*grade conversion*'. This problem does not appear in the Spider-Syn and ADV_{BERT} examples, and some $\text{ADV}_{\text{GLOVE}}$ examples do not have this problem, such as the second example in Table 5.6. Some examples reward the model for finding the schema item that is most similar to the question token, while others penalize this pattern, which causes the model to fail to learn. Thus the model with $\text{ADV}_{\text{GLOVE}}$ neither defends against $\text{ADV}_{\text{GLOVE}}$ attack nor even obtains good performance on the Spider.

Spider:	Which **courses** are taught on **days** MTW?
Spider-Syn:	Which **curriculum** are taught on days MTW?
$\text{ADV}_{\text{GLOVE}}$:	Which **trajectory** are taught on **jour** MTW ?
ADV_{BERT}:	Which **classes** are taught on **times** MTW ?
Spider:	Show the name and **phone** for **customers** with a **mailshot** with **outcome** code 'No Response'
Spider-Syn:	Show the name and **telephone** for **clients** with a mailshot with outcome code 'No Response'.
$\text{ADV}_{\text{GLOVE}}$:	Show the name and **telephones** for customers with a mailshot with outcome code 'No Response'.
ADV_{BERT}:	Show the name and **telephone** for customers with a **mailbox** with **result** code 'No Response'.

Table 5.6: Two questions in Spider with corresponding versions of Spider-Syn, $\text{ADV}_{\text{GLOVE}}$ and ADV_{BERT}.

5.3.5 Ablation Study

To analyze the individual contribution of our proposed techniques, we have run some additional experiments and show their results in Table 5.5. Specifically, we use RAT-SQL_B+SPR, RAT-SQL_B+SPR_{SYN}, RAT-SQL_B+$\text{SPR}_{\text{SPR\&SYN}}$, and RAT-SQL_B+ADV_{BERT} as base models, then we apply different schema annotation methods to these model and evaluate their performance in different development sets. Note that all base models use the Spider original schema annotations.

First, for all base models, we found that MAS consistently improves the model performance when questions are modified by synonym substitution. Specifically, when evaluating on Spider-Syn, using ManualMAS achieves the best performance, because the ManualMAS contains the synonym substitutions of Spider-Syn. Meanwhile, when evaluating on worst-case adversarial attacks, AutoMAS mostly outperforms ManualMAS. Considering that the AutoMAS is automatically generated, AutoMAS would be a simple and efficient way to improve the robustness of text-to-SQL models.

5.3.6 Further Discussion on MAS

ManualMAS utilizes the same synonym annotations on Spider-Syn, the same relationship as AutoMAS with $\text{ADV}_{\text{GLOVE}}$, and we design this mechanism to demonstrate the effectiveness of MAS in an ideal case. By showing the superior performance of ManualMAS on Spider-Syn, we confirm that the failure of existing models on Spider-Syn is largely because they rely on the lexical correspondence, and MAS improves the performance by repairing the lexical link. Besides, MAS has the following advantages:

Approach	Spider	Spider-Syn	$\text{ADV}_{\text{GLOVE}}$	ADV_{BERT}
GNN	**48.5%**	23.6%	25.4%	28.9%
GNN + ManualMAS	44.0%	**38.2%**	22.9%	26.2%
GNN + AutoMAS	44.0%	29.5%	**39.8%**	**31.8%**
IRNet	**53.2%**	28.4%	26.4%	29.0%
IRNet + ManualMAS	49.7%	**39.3%**	24.0%	27.2%
IRNet + AutoMAS	53.1%	35.1%	**44.3%**	**35.6%**

Table 5.7: Evaluation on the combination of MAS with GNN and IRNet respectively.

- Compared to adversarial training, MAS does not need any additional training. Therefore, by including different annotations for MAS, the same pre-trained model could be applied to application scenarios with different requirements of robustness to synonym substitutions.
- MAS could also be combined with existing defenses, e.g., on adversarially trained models, as shown in our evaluation.

We add the evaluation on the combination of MAS with GNN and IRNet respectively, shown in Table 5.7. The conclusions are similar to RAT-SQL: (1) MAS significantly improves the performance on Spider-Syn, and ManualMAS achieves the best performance. (2) AutoMAS also considerably improves the performance on adversarial attacks.

5.4 Summary

This chapter introduce Spider-Syn, a human-curated dataset based on the Spider benchmark for evaluating the robustness of text-to-SQL models for synonym substitution. We found that the performance of previous text-to-SQL models drop dramatically on Spider-Syn, as well as other adversarial attacks performing the synonym substitution. We design two categories of approaches to improve the model robustness, i.e., multi-anotation selection and adversarial training, and demonstrate the effectiveness of our approaches.

Chapter 6

Re-appraising the Schema Linking Mechanism in Text-to-SQL

In Chapter 5, we study the text-to-SQL models against synonym substitution on schema item words, and experiments show that the performance of existing models drops significantly. To address this challenge, we propose two methods to improve the robustness of the model. We believe that the research of the previous chapter is the destruction and repair of the schema linking mechanism. In this chapter, we further discuss the schema linking mechanism in text-to-SQL. For definitions of schema linking related terms, see Chapter 2.3.2.

Figure 6.1 presents an example of schema linking and the EMSL feature matrix. Most previous work relies on this exact lexical matching to obtain schema linking features. Following the work of [Krishnamurthy et al., 2017, Guo et al., 2019, Bogin et al., 2019a], EMSL is used in many subsequent works [Wang et al., 2020, Cai et al., 2021, Xu et al., 2021, Lei et al., 2020, Yu et al., 2021, Shi et al., 2021] and has been shown to be effective. For example, the ablation study in [Guo et al., 2019] shows that removing the schema linking module incurs the most significant performance decrease.

Although EMSL has been widely used and helps models obtain the state-of-the-art performance on some text-to-SQL benchmarks [Yu et al., 2018b, Zhong et al., 2017], in this chapter, we show that EMSL renders models vulnerable to noise in the input, particularly synonym substitution and typos. We then investigate whether text-to-SQL models can preserve good prediction performance without EMSL. Previous ablation studies [Guo et al., 2019, Wang et al., 2020] claiming the necessity of the schema linking module were conducted without pretrained language models (PLMs) such as BERT. In fact, we find that when a pretrained language model is used, removing EMSL has very little impact on the performance of the model. This observation is consistent for different model architectures and training schemes, such as RATSQL [Wang et al., 2020], GNN [Bogin et al., 2019a], and GAP [Shi et al., 2021].

We evaluate the models in three settings: the original Spider benchmark without input noise [Yu et al., 2018b], synonym substitution [Gan et al., 2021a], and a new typo injection setting. Results show that the use of a pretrained language model can provide the same performance benefit as EMSL, while achieving better robustness against synonym substitution and typos. Removing EMSL also allows the model to obtain better results when training with synonym substitution samples. We also show that MAS (Multi-Annotation Selection) [Gan et al., 2021a], a method designed to improve model robustness with EMSL, can also improve models without EMSL. In conclusion, we demonstrate that with pretrained language models, EMSL is no longer a necessary building block of text-to-SQL models.

The EMSL and pretrained language models work for the schema linking module that learns a schema linking score to decide which schema item to be selected. There are two design choices for this score: the first sees it as a direct relation between the question and schema items [Bogin et al., 2019a]; while the second considers the question and schema items together [Wang et al., 2020]. Experiments show that the first design is more interpretable and can improve the performance of the RATSQL+GAP+NatSQL [Gan et al., 2021c] model close to the state-of-the-art model.

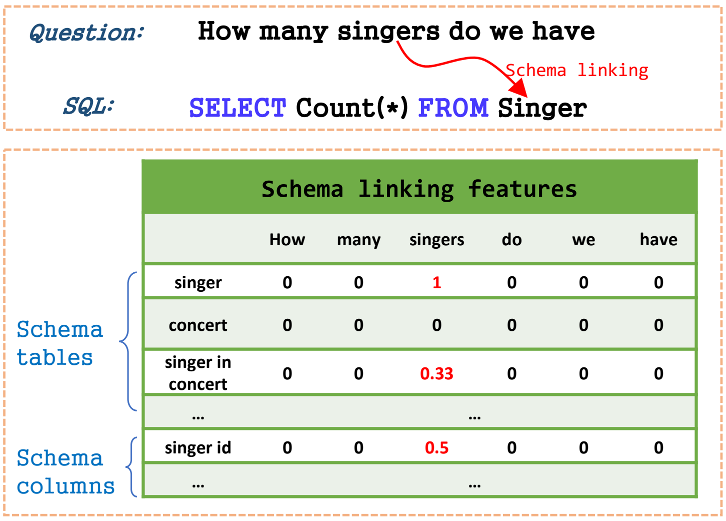
6.1 Schema Linking

Following SQLNet [Xu et al., 2017], most text-to-SQL models generate the SQL structure first, and then fill in the schema items [Gan et al., 2020]. Schema linking is needed in this workflow to locate the schema items from the question. Prior works show that models without schema linking perform poorly on text-to-SQL tasks, such as the sequence-to-sequence model [Yu et al., 2018b].[1]

6.1.1 Schema Linking Feature

Figure 6.1 presents an example of schema linking features. The word *'singers'* in the question exactly matches (modulo stemming) the schema table name *'singer'*, giving feature value 1. It does not match the table *'concert'*, giving

[1]Note that prior works often use the phrase *schema linking* in different ways; it may refer to the schema linking feature or module or both.


Question: How many singers do we have
Schema linking
SQL: SELECT Count(*) FROM Singer
Schema tables
Schema columns

Schema linking features

	How	many	singers	do	we	have
singer	0	0	1	0	0	0
concert	0	0	0	0	0	0
singer in concert	0	0	0.33	0	0	0
…			…			
singer id	0	0	0.5	0	0	0
…			…			

Figure 6.1: An example of schema linking and exact match based schema linking (EMSL) feature matrix.

value 0; and matches one of the three words in *'singer in concert'*, giving value 0.33. This type of schema linking feature (EMSL) based on exact lexical matching is the most common [Guo et al., 2019, Bogin et al., 2019a, Wang et al., 2020, Cai et al., 2021, Xu et al., 2021, Lei et al., 2020, Yu et al., 2021, Shi et al., 2021]. Some papers may not mention this exact matching explicitly, but it can be found in their published code. Implementation details vary; for example, some works add ConceptNet [Speer and Havasi, 2012a] to get more linking features [Guo et al., 2019, Tan et al., 2021].

EMSL is often taken to be essential: ablation studies show that removing EMSL causes the biggest performance decline compared to removing other removable modules [Guo et al., 2019, Wang et al., 2020]. Wang et al. [2020] consider that the representations produced by vanilla self-attention were insensitive to textual matches even though their initial representations were identical, i.e., the EMSL is needed for textual matches. However, we argue that a well-designed encoder can solve this problem, and note that the feature values in Figure 6.1 are equal to the average dot product results when using lemma one-hot embeddings, which means a proper embedding can replace EMSL. We discuss details in Section 6.2.3.

6.1.2 Schema Linking Module

We believe that a text-to-SQL model with good performance can ignore the schema linking feature, but it must include a schema linking module. While implementation details of such models differ, the common factor is the calculation of a similarity score between each question word and schema item: correct schema items should obtain higher similarity scores.

One difficulty in calculating the similarity scores is how to represent a schema item containing multiple words. For example, we need a proper vector to represent the *singer in concert* table in Figure 6.1, so that it has a higher score when calculating similarity with the words *singer* or *concert* in a question. If we cannot find such a vector, we need EMSL as the similarity score, e.g., use the 0.33 in Figure 6.1 to represent the similarity between *singer in concert* table and word *singer* in the question.

Schema linking modules output an attention score from computing the schema linking feature and word embeddings. There are currently two attention mechanism designs. The first calculates a score that relates the question on one side, to the schema items on the other; this assumes that all the schema information we need is available in the question. This second approach assumes the opposite: as schema items may be implicit [Guo et al., 2019], the attention between one schema item and others is needed, and it therefore takes question and schema items together as input to produce the score. Section 6.3.6 discusses the differences between the two designs in the form of illustration.

6.2 Case Study

In this section, we conduct an ablation study on EMSL using different models, including GNN [Bogin et al., 2019a], IRNet [Guo et al., 2019], and RAT-SQL [Wang et al., 2020]. We then conduct a more detailed examination using RATSQL, which is the most competitive model architecture. We did not choose the SOTA models [Cao et al., 2021, Hui et al., 2022] in our case study because they build a graph relying on the EMSL. Removing the EMSL will break their graph neural networks, which is one of their main contributions.

6.2.1 Ablation Study on EMSL

Table 6.1 presents the ablation study results of three base models. The results of RATSQL here are different from that of [Wang et al., 2020] because Wang et al. [2020] remove the cell value linking first and then EMSL. According to the magnitude of the decline, our results are similar to theirs. According to [Wang et al., 2020, Guo et al., 2019], they observe the biggest performance degradation by removing EMSL. Since then, EMSL has become a necessary module for most researchers to build text-to-SQL models.

We want to challenge this view and carry out the comparative experiment

Model	Exact Match Acc
GNN	47.6%
GNN **w/o** EMSL	24.9%
IRNet	48.5%
IRNet **w/o** EMSL	40.5%
RATSQL	62.7%
RATSQL **w/o** EMSL	51.9%

Table 6.1: Accuracy of three based models ablations on the development set. EMSL means schema linking feature based on the exact lexical match. The IRNet results are copied from the original paper [Guo et al., 2019], while others are conducted by ourselves.

Model	Exact Match Acc
GNN+BERT	49.3%
GNN+BERT **w/o** EMSL	47.1%
RATSQL+BERT	69.7%
RATSQL+BERT **w/o** EMSL	69.3%
RATSQL+GAP	71.8%
RATSQL+GAP **w/o** EMSL	71.7%

Table 6.2: Accuracy of three models with PLM ablations on the development set. The GAP [Shi et al., 2021] is a pretrained model based on RoBERTa [Liu et al., 2019]

in Table 6.2. Comparing Table 6.1 and Table 6.2, it can be found that PLMs compensate for the function of EMSL, i.e., the performance in Table 6.2 is less degraded than that in Table 6.1 after removing EMSL.

From another perspective, BERT and its subsequent pretrained language model significantly improve the performance of models that do not use EMSL, which explains why some models can achieve higher performance improvements through BERT. For example, EditSQL [Zhang et al., 2019] does not use EMSL, while it obtains the highest performance improvement by extending BERT, as shown on the Spider leaderboard [2].

6.2.2 BERT vs GLOVE

The base RATSQL uses GLOVE [Pennington et al., 2014] for word embedding. There are two main reasons why BERT [Devlin et al., 2019] is better than GLOVE at schema linking. The first reason is that BERT can better deal with out-of-vocabulary words. BERT converts these words into subwords, so BERT makes sure different word is represented by a unique vector. However, GLOVE

[2]`https://yale-lily.github.io/spider`

cannot handle out of vocabulary words. Researchers generally replace them with a custom unknown (UNK) word vector. Suppose there are multiple words outside the GLOVE vocabulary in one schema. In that case, it is equivalent to multiple schema items being annotated as UNK, which will cause the model without EMSL to be unable to distinguish different schema items due to the same word vector.

The second reason is that GLOVE is not as good as BERT in the face of schema items containing multi-words. As opposed to static embeddings provided by GLOVE, BERT provides dynamic lexical representations generated by analyzing the context. Take the *bandmate id* column in the Spider dataset as an example. The cosine of the vectors for the two words *bandmate* and *id* in GLOVE is negative, which means if we sum these two vectors together to represent the *bandmate id* column, the sum vector will inevitably lose some information. The word vector output by BERT is calculated based on the context, so although adjacent words may be unrelated in word meaning, their word vectors will still be highly correlated. Figure 6.2, generated by the bertviz [Vig, 2019], presents the BERT head view of attention patterns in the one transformer layer where the word *bandmate* clearly links to the word *id*.

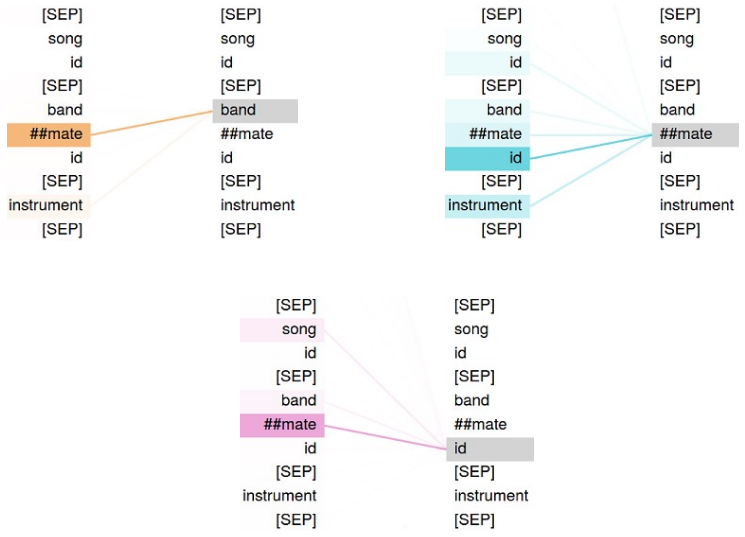
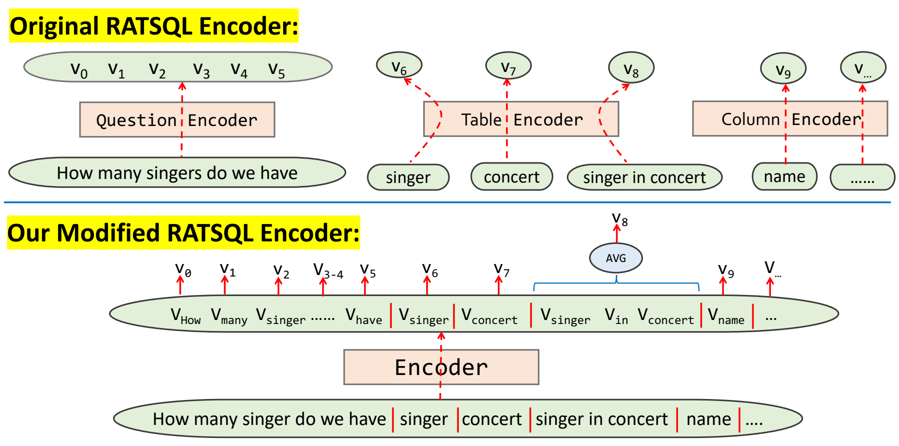


Figure 6.2: The BERT head view of attention patterns of word *bandmate* and *id* in the one transformer layer.

6.2.3 RATSQL Encoder

The text-to-SQL encoder is part of the schema linking module. As discussed in Section 6.1.2, we expect that the correct schema item vectors obtained from the



Original RATSQL Encoder:
v0 v1 v2 v3 v4 v5
Question Encoder
How many singers do we have
v6
v7
v8
Table Encoder
singer
concert
singer in concert
v9
v...
Column Encoder
name
......
Our Modified RATSQL Encoder:
v8
AVG
v0 v1 v2 v3-4 v5 v6 v7 v9 v...
VHow Vmany Vsinger ...... Vhave | Vsinger | Vconcert | Vsinger Vin Vconcert | Vname | ...
Encoder
How many singer do we have | singer | concert | singer in concert | name | ....

Figure 6.3: The original RATSQL encoder structure and our modified version.

encoder are as close to the question vector as possible. The SQL cares about which schema item to use instead of the words in the schema item. Therefore, unlike keeping every question word vector, only one vector is used to present the schema item even if it contains multiple words. Since both the encoder mechanics and content style are different between question and schema, RATSQL uses different encoders to encode the question and schema separately, as shown in the upper part of Figure 6.3. These three encoders are based on biLSTM and have similar structure and size.

We believe that the shortcoming of the original RATSQL design is the use of three encoders. For example, in the initial state, the parameters of the three encoders are different. Therefore, even though the word *'singers'* appears in the question, the vector v_6 initially generated by the table encoder is probably irrelevant to all vectors output by the sentence encoder. It does not matter when using EMSL for both training and evaluation because we can link the v_6 to v_2 through EMSL. However, when without EMSL, it requires the v_6 from the table encoder must close to the vectors from the question encoder, which is more challenging to train than using only one encoder, as shown in the lower part of Figure 6.3. Since the output of our modification is the same as the original, it can be easily replaced and connected to the subsequent modules.

In the lower part of Figure 6.3, our modification is inspired by several text-to-SQL models with BERT, including RATSQL+BERT [Wang et al., 2020, Guo et al., 2019, Zhang et al., 2019]. In our modification, RATSQL uses only the BERT encoder instead of the three encoders. We believe using three encoders is one of the main reasons why the base RATSQL performance significantly drops when removing EMSL. For the convenience of discussion, we named our modified RATSQL as RATSQL$_O$, where O means one encoder.

RATSQL$_O$ uses only one encoder whose structure and size are the same as the original question encoder. For the schema item representation, RATSQL takes the hidden state after all the words of the entire schema item are encoded, while RATSQL$_O$ takes the average of all word encodings. The advantage of the RATSQL$_O$ is that v_6, v_8, and v_2 initially have a certain similarity, which benefits the schema linking in both single and multi words. Besides, RATSQL$_O$ deal with words outside the GLOVE vocabulary better than RATSQL. Suppose the word *concert* and *stadium* are outside the GLOVE vocabulary, the v_7 and v_9 output from RATSQL table encoder will be the same since their inputs are the same UNK vector. However, the RATSQL$_O$ encoder (BiLSTM) output different vectors for v_7 and v_9 because the contents before and after the word *concert* and *stadium* are different. In this way, even if there are multiple UNK words, the RATSQL$_O$ encoding vector will be different.

Expressed in simpler terms, an encoder maps an input vector to a new vector space based on non-linear changes. Therefore, the v_2 and v_8 in Figure 6.3 would be significantly different if outputted from different encoders, making them hard to obtain a high similarity score. In RATSQL$_O$, the two v_singer are generated by the same encoder, so they are inherently similar. Besides, the v_8 in RATSQL$_O$ is calculated by an average function as linear change, so v_8 can keep similar to the v_2. The BERT is based on a unique encoder similar to RATSQL$_O$, which makes it easier to construct the schema linking without the EMSL.

6.3 Experiment

6.3.1 Generating Typos

To evaluate robustness against typos, we randomly insert a letter into the correct schema annotation word. (This is enough to break EMSL, so we do not also modify the question words). We generated three typo development sets, named Spider-T1 to Spider-T3. The typos in Spider-T1 are generated by randomly inserting a letter at any position except the end. In contrast, Spider-T2 appends a random letter at the end of the schema annotation words. We examine these separately: the BERT tokenizer may be able to split Spider-T2 typos into a correct word and a suffix, but is less likely to split the Spider-T1 typos well. We convert every schema annotation word in Spider-T1 and T2 to typos when word length is greater than five letters; typos are generally more likely to occur in longer words, and words with more than five letters account for about 40% of the dataset. Spider-T3 is then the same as Spider-T1, but only converts the most frequent schema item words to typos. While Spider-T1 and T2 simulate

	Number of errors			Number of example with errors		
Approach	**Multi words**	**Single word**	**UNK word**	**Multi words**	**Single word**	**UNK word**
RATSQL	118	57	13	112 (10.8%)	54 (5.2%)	12 (1.2%)
RATSQL **w/o** EMSL	178	107	33	170 (16.4%)	93 (9.0%)	30 (2.9%)
RATSQL$_O$	136	51	11	125 (12.1%)	50 (4.8%)	11 (1.1%)
RATSQL$_O$ **w/o** EMSL	152	63	15	141 (13.6%)	59 (5.7%)	14 (1.4%)
RATSQL$_B$	55	38	-	53 (5.1%)	37 (3.6%)	-
RATSQL$_B$ **w/o** EMSL	65	34	-	65 (6.3%)	34 (3.3%)	-

Table 6.3: Statistics of the types of error column predictions of different models evaluated on the Spider development set. The larger the number, the worse the performance.

the impact of large numbers of typos in extreme cases, Spider-T3 evaluates the impact of a more realistic, smaller number of typos.

Model	Spider
RATSQL	62.7%
RATSQL **w/o** EMSL	51.9%
RATSQL$_O$	62.2%
RATSQL$_O$ **w/o** EMSL	58.4%

Table 6.4: Accuracy of two RATSQL ablations on the development set.

6.3.2 Experimental Setup

We evaluate the previous state-of-the-art models on Spider [Yu et al., 2018b], Spider-T, and Spider-Syn [Gan et al., 2021a] datasets. All experiments were performed on a machine with an Intel i5 9600 3.1GHz processor and a 24GB RTX3090 GPU. Since the Spider test set is not publicly accessible and Spider-Syn and Spider-T do not contain test sets, our evaluation is based on the development sets. The Spider-Syn benchmark contains three development sets: Spider-Syn, ADV$_{\text{BERT}}$, and ADV$_{\text{GLOVE}}$, for evaluating model robustness against synonym substitution. Therefore, we have the following evaluation sets:

- **Spider**: The original Spider development set with 1,034 examples.
- **Spider-T1, T2 and T3**: Three typo development sets with 1,034 examples respectively, discussed in Section 6.3.1.
- **Spider-Syn**: The human-curated development set built upon Spider, for evaluating synonym substitution in real-world question paraphrases.
- **ADV$_{\text{BERT}}$:** The set of adversarial examples generated by BERT-Attack [Li et al., 2020].
- **ADV$_{\text{GLOVE}}$:** The set of adversarial examples generated using the nearest GLOVE word vector [Pennington et al., 2014, Mrkšić et al., 2016].

Approach	**Spider**	**Spider-T1**	**Spider-T2**	**Spider-T3**	**Spider-Syn**	**ADV$_{\text{GLOVE}}$**	**ADV$_{\text{BERT}}$**
RATSQL	**62.7%**	**23.9%**	**26.4%**	**51.2%**	33.9%	30.9%	37.1%
RATSQL **w/o** EMSL	51.9%	20.8%	21.7%	44.1%	**39.1%**	**38.1%**	**40.9%**
RATSQL$_O$	**62.2%**	**22.8%**	**25.7%**	**51.6%**	32.1%	32.7%	36.3%
RATSQL$_O$ **w/o** EMSL	58.4%	20.8%	23.3%	51.5%	**42.6%**	**38.6%**	**43.8%**
RATSQL$_B$	**69.7%**	30.9%	54.8%	**63.2%**	48.2%	38.0%	48.8%
RATSQL$_B$ **w/o** EMSL	69.3%	**32.3%**	**66.2%**	63.0%	**52.7%**	**45.4%**	**54.3%**
RATSQL$_{BS}$	68.1%	33.6%	58.1%	62.7%	58.0%	47.7%	55.7%
RATSQL$_{BS}$ **w/o** EMSL	**69.7%**	**38.1%**	**66.4%**	**65.0%**	**60.4%**	**51.0%**	**58.8%**
RATSQL$_G$	**71.8%**	48.1%	64.6%	68.0%	54.6%	46.6%	54.8%
RATSQL$_G$ **w/o** EMSL	71.7%	**53.4%**	**67.6%**	**68.6%**	**58.7%**	**49.4%**	**57.3%**

Table 6.5: Exact match accuracy on original (Spider), typos (Spider-T1 to T3), and synonym substitution (Spider-Syn, ADV$_{\text{GLOVE}}$, and ADV$_{\text{BERT}}$) development sets.

Our evaluation is based on the exact match metric defined in the original Spider benchmark. This metric measures whether the syntax tree of the predicted query without condition values is the same as that of the gold query. Our experiment setting is consistent with the ablation study in Section 6.2.1. Following the case study in Section 6.2, we evaluate different variants of the RATSQL model:

- **RATSQL**: The base RATSQL+GLOVE model trained on Spider using EMSL in training and evaluation [Wang et al., 2020].
- **RATSQL$_\text{O}$**: Our modified RATSQL+GLOVE model trained on Spider using EMSL in training and evaluation, discussed in Section 6.2.3.
- **RATSQL$_\text{B}$**: The RATSQL+BERT model trained on Spider using EMSL in training and evaluation. (Note that RATSQL$_O$+BERT is just RATSQL+BERT: using BERT means that the BERT encoder will replace all encoders in Figure 6.3).
- **RATSQL$_\text{BS}$**: RATSQL+BERT trained on **Spider-Syn** using EMSL [Gan et al., 2021a].
- **RATSQL$_\text{G}$**: RATSQL+GAP trained on Spider using EMSL [Shi et al., 2021].
- **w/o EMSL:** Models do not use EMSL in training and evaluation, consistent with Tables 6.1 and 6.2.
- **ManualMAS** [Gan et al., 2021a]: Schema annotations include synonyms used in Spider-Syn.
- **AutoMAS** [Gan et al., 2021a]: Schema annotations include synonyms generated according to the nearest GLOVE word vector.

Approach	Spider	Spider-Syn	ADV$_{\text{GLOVE}}$	ADV$_{\text{BERT}}$
RATSQL$_B$ + ManualMAS	67.4%	**62.6%**	34.2%	44.5%
RATSQL$_B$ + ManualMAS **w/o** EMSL	**68.6%**	58.9%	**43.6%**	**53.1%**
RATSQL$_B$ + AutoMAS	68.7%	**56.0%**	61.2%	52.5%
RATSQL$_B$ + AutoMAS **w/o** EMSL	**68.9%**	55.3%	**62.1%**	**54.7%**
RATSQL$_{BS}$ + ManualMAS	65.6%	59.5%	46.9%	51.7%
RATSQL$_{BS}$ + ManualMAS **w/o** EMSL	**68.7%**	**61.7%**	**50.3%**	**58.8%**
RATSQL$_{BS}$ + AutoMAS	66.8%	57.5%	61.0%	55.7%
RATSQL$_{BS}$ + AutoMAS **w/o** EMSL	**69.2%**	**59.4%**	**63.2%**	**59.0%**

Table 6.6: Evaluation on the combination of MAS with RATSQL$_B$ and RATSQL$_{BS}$ respectively.

6.3.3 Evaluation on Spider

Table 6.4 presents the exact matching accuracy of models trained on the Spider training set. It is clear that our RATSQL$_O$ significantly improves the without-EMSL performance. Tables 6.4 and 6.2 illustrate that the EMSL can be replaced by better encoding. The performance of RATSQL is slightly better than that of RATSQL$_O$, because Guo et al. [2019] conducted 100 time hyperparameter search to optimize the RATSQL while we did not do that. Therefore, when we modify the model structure, it may cause a slight performance degradation.

Error Analysis Table 6.3 presents the error type statistics in the error column prediction. We count the prediction errors of single words, multiple words, and words outside the GLOVE vocabulary (UNK word) when the predicted SQL structure is correct. As BERT does not share GLOVE's vocabulary limitations, the UNK entry for RATSQL$_B$ is empty. Random initialization means that model results after each training may vary slightly, so we only focus on the more salient features.

Although the results of RATSQL and RATSQL$_O$ are similar, RATSQL$_O$ consistently outperforms RATSQL in three error types when EMSL is removed; this supports the view we discuss in Section 6.2.3. More importantly, the single-word performance of RATSQL$_O$ without EMSL is close to that of RATSQL and RATSQL$_O$. As discussed in Section 6.2.2, the representation ability on multi-word of GLOVE is worse than that of BERT. The results support this view where the performance of RATSQL$_O$ and RATSQL on multi-word is worse than that on single-word. When replacing the GLOVE with BERT, due to the improvement of its multi-word representation ability, the performance of RATSQL$_B$ with and without EMSL are close in single and multiple words. From the right side of Table 6.3, it can also be found that the BERT brings around 5% absolute improvement on multi-word, while that on single-word is only 2%.

6.3.4 Robustness Evaluation

Typo Results Table 6.5 presents the robustness evaluation results on several datasets. For typos, GLOVE will treat them as UNK words, so the RATSQL and RATSQL$_O$ cannot obtain good performance on Spider-T1 and T2 due to too many UNK words. The RATSQL$_O$ without EMSL significantly outperforms the RATSQL without EMSL in Spider-T3, which is another evidence that the RATSQL$_O$ is better in handling UNK words. After using PLMs, the performance on typos has been significantly improved, especially on Spider-T2. Spider-T3 contains only a few typos, i.e., it is close to the Spider to some extent. Thus, the T3 result characteristics are close to Spider, i.e., their performance gap between with and without EMSL is close. With the increase of typos, the performance gap will be expanded, where the model+PLM without EMSL will be better.

More Typos Besides generating typos by inserting a letter, we also generate typos by deleting a letter and swapping the letter position, named the generated development set Spider-T4 and Spider-T5, respectively. Like Spider-T1 and T2, here we only convert the words whose length is greater than five letters to typos. Table 6.7 presents the exact match accuracy on Spider-T4 and Spider-T5 development sets. Since PLM handles typos in Spider-T4 and T5 similar to Spider-T1, their evaluation results are also similar. Besides, we observe that the results of models using GLOVE in Spider-T4 are the best, followed by in T5, then in T2, and finally in T1. To understand this phenomenon, we found that although the number of generated typos is the same among these datasets, Spider-T1 has the most GLOVE UNK words, followed by T2, then T5, and T4 contains the least UNK words. It can be seen that in the case of fewer UNK words, the model+GLOVE can generate better encoding so that the model+GLOVE without EMSL surpasses that with EMSL in Spider-T4 and T5.

Synonym Substitution Results Gan et al. [2021a] propose three development sets for evaluating the robustness of text-to-SQL models against synonym substitution, including: Spider-Syn, ADV$_{\text{BERT}}$, and ADV$_{\text{GLOVE}}$. Table 6.5 shows that models without EMSL consistently outperform those with EMSL when evaluated against Spider-Syn, ADV$_{\text{GLOVE}}$ and ADV$_{\text{BERT}}$. When using PLMs, RATSQL$_B$ and RATSQL$_G$ without EMSL show a huge performance improvement on these three development sets with only a tiny performance loss on Spider. RATSQL$_O$ without EMSL consistently outperforms RATSQL without EMSL, which means a reasonable design can reduce reliance on EMSL. Unlike

Approach	**Spider-T4**	**Spider-T5**
RATSQL	29.0%	28.6%
RATSQL **w/o** EMSL	**32.8%**	**30.1%**
RATSQL$_O$	27.6%	26.5%
RATSQL$_O$ **w/o** EMSL	**34.5%**	**31.2%**
RATSQL$_B$	34.9%	32.6%
RATSQL$_B$ **w/o** EMSL	**38.8%**	**35.0%**
RATSQL$_{BS}$	35.6%	32.6%
RATSQL$_{BS}$ **w/o** EMSL	**40.3%**	**38.2%**
RATSQL$_G$	46.7%	46.8%
RATSQL$_G$ **w/o** EMSL	**50.6%**	**50.7%**

Table 6.7: Exact match accuracy on Spider-T4 and Spider-T5 development sets.

other models, the RATSQL$_{BS}$ without EMSL outperforms that with EMSL in all evaluation sets.

MAS Results Gan et al. [2021a] also propose a MAS method to improve the robustness of text-to-SQL models. MAS provides multiple annotations to repair the breaking of EMSL due to synonym substitutions. Although we advocate not relying on EMSL, MAS can still improve the performance of models without EMSL, as shown in Table 6.6. Comparing the data in Table 6.5 and Table 6.6, ManualMAS improves the performance of RATSQL$_B$ and RATSQL$_{BS}$ with and without EMSL on Spider-Syn development set since the ManualMAS provide synonym annotations appearing in the Spider-Syn. In the same way, AutoMAS has also improved their performance on ADV$_{\text{GLOVE}}$. Experimental results show that although MAS is designed to repair EMSL, it is still effective for models without EMSL. Besides, based on MAS, the overall performance of the model without EMSL is still better than that with EMSL. In general, even though EMSL is not used, a reasonable annotation is still essential to the text-to-SQL problem.

6.3.5 Further Discussion on EMSL

The text-to-SQL model can quickly locate the correct schema items through EMSL, but this advantage will cause the models to not work properly when EMSL fails. To better understand the impact of EMSL on text-to-SQL models, we present the question-table attention [3] extracted from RATSQL$_B$ with and without EMSL in Figure 6.4. In the first example, we can see that the alignment

[3]It is named *m2t_align_mat* in the code: `https://github.com/microsoft/rat-sql/blob/master/ratsql/models/spider/spider_enc_modules.py`

Que:	For each stadium , how many concerts play there ?
Error:	Failed to predict the implicit *'stadium name'* column.
Que:	What is the degree summary name that has the most number of students enrolled ?
Error:	Generate a redundant *'student enrolment courses'* table.
Que:	Which language is the most popular on the Asia continent ?
Error:	Failed to predict the implicit *'country'* table.

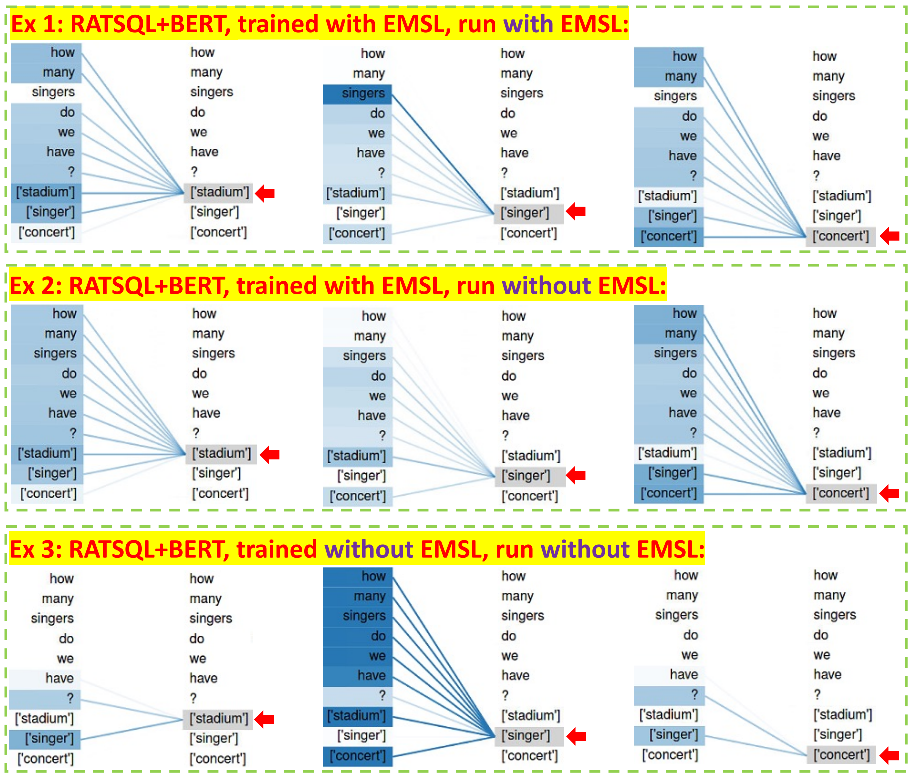
Table 6.8: These error predictions are outputted from the RATSQL$_G$ with the first design choice while using the original (2nd) design choice can predict correctly.

Model	Spider
RATSQL^1_G	70.2%
RATSQL^2_G	71.8%
RATSQL^2_G **with** NatSQL [Gan et al., 2021c]	73.7%
LGESQL + ELECTRA [Cao et al., 2021]	75.1%
RATSQL^1_G **with** NatSQL	75.5%
S2SQL [Hui et al., 2022] **(Current SOTA)**	76.4%

Table 6.9: Accuracy of RATSQL$_G$ with different schema linking design choices on the Spider development set. The superscript number of RATSQL$_G$ indicate the design choice. The results are compared with the top two models in the Spider leaderboard.

score between table *singer* and question word *singer* is the biggest, while we can not observe a clear connection between other tables and question word *singer*. However, when removing the EMSL in the second example, the alignment score between table *singer* and question word *singer* drop clearly, and the connection between other tables and question word *singer* becomes clear. It can be seen that under other conditions unchanged, only removing EMSL has a considerable impact on the model trained with EMSL.

The third example is extracted from RATSQL$_B$ without EMSL. Different from the RATSQL$_B$ with EMSL, the *singer* table has a high alignment score not only with the word *singer* but also with the whole sentence. Since the loss function only calculates whether the output schema items are correct, the model does not care which question word the correct schema item is linked to. Therefore, the attention of the RATSQL$_B$ without EMSL is quite different from that with EMSL. The significant difference of the trained models may be one of the reasons why the overall performance of RATSQL$_{BS}$ without EMSL is better than that with EMSL. Because the training data in RATSQL$_{BS}$ contain many synonym substitution examples, and these examples do not have EMSL features, it requires the model to find a balance between states shown in examples 1 and



Ex 1: RATSQL+BERT, trained with EMSL, run with EMSL:
Ex 2: RATSQL+BERT, trained with EMSL, run without EMSL:
Ex 3: RATSQL+BERT, trained without EMSL, run without EMSL:
how
many
singers
do
we
have
?
['stadium']
['singer']
['concert']

Figure 6.4: Examples of the question-table attention. The darker the color, the greater the attention score. The first two examples are extracted from RATSQL$_B$, while the last one is from RATSQL$_B$ without EMSL. Each attention subgraph represents the attention between only one table schema and other words.

3 of Figure 6.4, which increases the difficulty of training.

6.3.6 Schema Linking Module Design Choices

As discussed in Section 6.1.2, there are two design choices for the schema linking module, where the original RATSQL chose the second one. We modify the RAT-SQL schema linking module according to the first design, which observes that the performance has dropped slightly. Error analysis shows that the RATSQL with the first design tends to use the schema items mentioned in the question and is not so good at dealing with the implicit schema items. Table 6.8 presents some error predictions from RATSQL$_G$ with the first schema linking module design.

Although the performance of the first design choice is slightly worse, we found that its schema linking performance is not inferior. The accuracy of the *SELECT* clause is the best way to measure the schema linking performance because every SQL contains at least one *SELECT* clause that only contains schema items. The *SELECT* accuracy of the first design choice is slightly (0.4%) better than the second one, which inspired us that the first one is likely to perform well if removing the implicit schema items. Fortunately, we found NatSQL, an SQL intermediate representation, that removes many implicit schema items from the SQL, making it closer to the natural language [Gan et al., 2021c]. Experiments show that the performance of RATSQL$_G$+NatSQL using the first design is better than using the second one. Table 6.9 gives a detailed performance comparison, from which it can be found that by replacing the design, the RATSQL$_G$+NatSQL is improved to the second place, evaluated on the development set, which is close to the current SOTA model. It should be noted that RATSQL$_G$+NatSQL does not use the complex graph neural network as S2SQL and LGESQL, nor does it use the ELECTRA, which is shown to be better than GAP [Clark et al., 2020, Cao et al., 2021, Hui et al., 2022]. In particular, S2SQL is a text-to-SQL model injecting syntax to a question-schema graph encoder for text-to-SQL parsing. LGESQL is also a text-to-SQL model employing the 1-hop edge features with a line graph in text-to-SQL. ELECTRA is a pre-trained language model similar to the BERT. Unlike the masked-language modeling (MLM) pre-training in BERT, ELECTRA is pre-trained based on 're-placed token detection' and obtains better performance than BERT in many tasks.

In addition, the first design is more interpretable because it would cause the model to infer the correct schema item from the question. So, you can see which part of the question is related to each selected schema item. Based on the second design, it is sometimes difficult to explain why the specific schema item is chosen due to the presence of some other schema items.

Attention Visualization of Different Schema linking Module Design Choices Figure 6.5 presents the attention weight of schema tables and illustrates why the first design choice is more interpretable. The SQL for the question in Figure 6.5 is '*SELECT* T1.City *FROM* Airports *AS* T1 *JOIN* Flights *AS* T2 *ON* T1.AirportCode = T2.DestAirport *GROUP BY* T1.City *ORDER BY* count(*) *DESC LIMIT* 1'. So, the table 'airports' and 'flights' are needed. Although models under both design choices predict this example correctly, their attention scores are quite different. We observe that the attention under the first design can locate the proper question words. However, the attention of the table 'flights' can not locate any question words when using the second design choice, which is difficult to explain why the 'flights' table was selected instead of the 'airlines' table with similar attention. It should be noted that the 'flights' table is mentioned implicitly, but it does not prevent the first design choice from

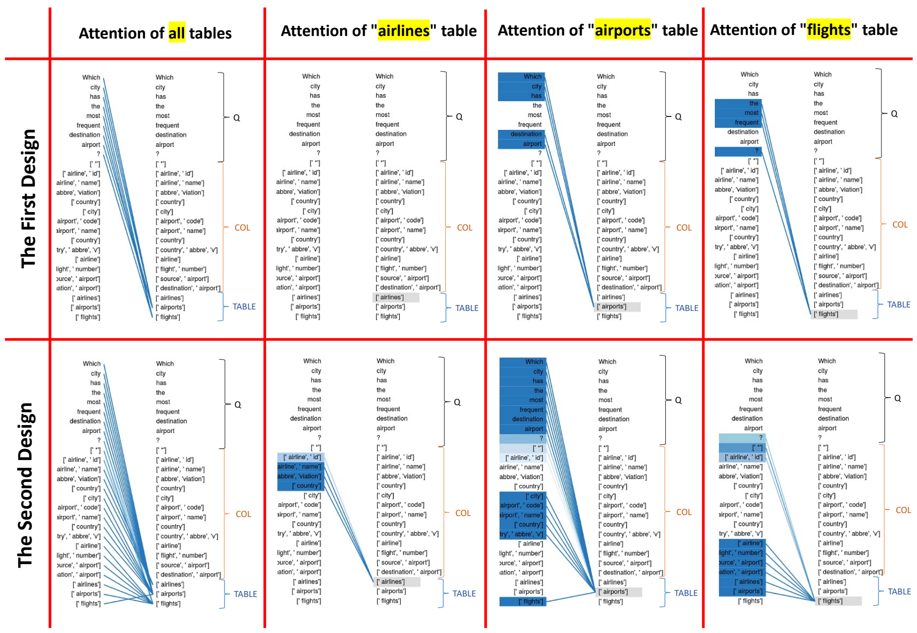


Figure 6.5: The attention weight of the schema table under different schema linking module design choices. The darker the color, the greater the attention score.

giving it the proper attention.

6.4 Summary

In this chapter, we study the mechanism of schema linking in detail and improve the model robustness, performance, and interpretability based on a better understanding of schema linking. Specifically, we demonstrate that with the presence of pretrained language models, EMSL is no longer a necessary building block to ensure a high performance on text-to-SQL benchmarks. We observe that when EMSL is used, models become overly reliant on it, making them vulnerable to attacks that break the exact-match assumptions of EMSL. On the other hand, we study the different schema linking module designs and found that a direct relation between the question and schema items is more interpretable and works well with intermediate representation SQL.

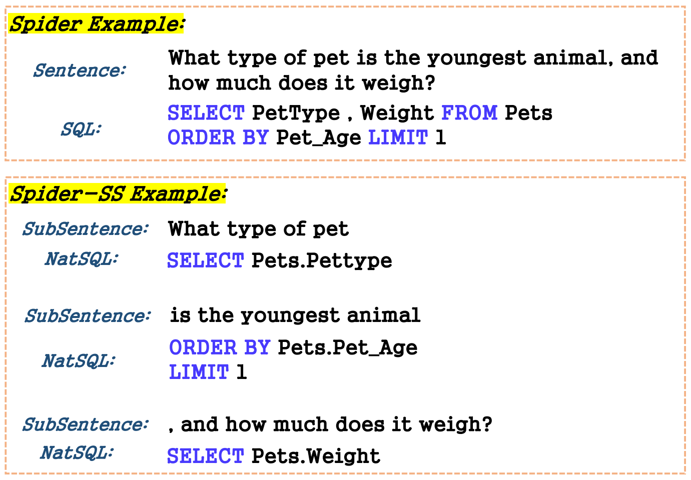
Chapter 7

Text-to-SQL Robustness against Compositional Generalization

In Chapter 6, we study the robustness and interpretability of schema linking. Besides the schema linking, compositional generalization also affects the robustness of the neural text-to-SQL models. Neural models in supervised learning settings show good performance on data drawn from the training distribution. However, generalization performance can be poor on out-of-distribution (OOD) samples [Finegan-Dollak et al., 2018, Suhr et al., 2020, Kaushik et al., 2020, Sagawa et al., 2020]. It might be the case even when the new samples are composed of known constituents; e.g., on the SCAN dataset [Lake and Baroni, 2018], many models give incorrect predictions for the input "jump twice and walk", even when "jump twice", "walk", and "walk twice" are seen during training. This (often lacking) ability to generalize to novel combinations of elements observed during training is referred to as *compositional generalization*.

Previous work on compositional generalization in text-to-SQL focuses on query split. For example, Shaw et al. [2021b] propose TMCD split based on SQL atoms and compounds analysis and question split based on length. Finegan-Dollak et al. [2018] proposes a query template-based split with word substitution that was much more challenging than the question split. However, these splits are limited by the dataset content, making it difficult to construct a challenging benchmark while ensuring that every question phrase (sub-sentence) appears in the training set.

Previous works [Chen et al., 2020b, Wang et al., 2021, Liu et al., 2020] improve generalization by enhancing the model's component awareness. Similarly,


Spider Example:

Sentence:	**What type of pet is the youngest animal, and how much does it weigh?**
SQL:	`SELECT PetType , Weight FROM Pets ORDER BY Pet_Age LIMIT 1`

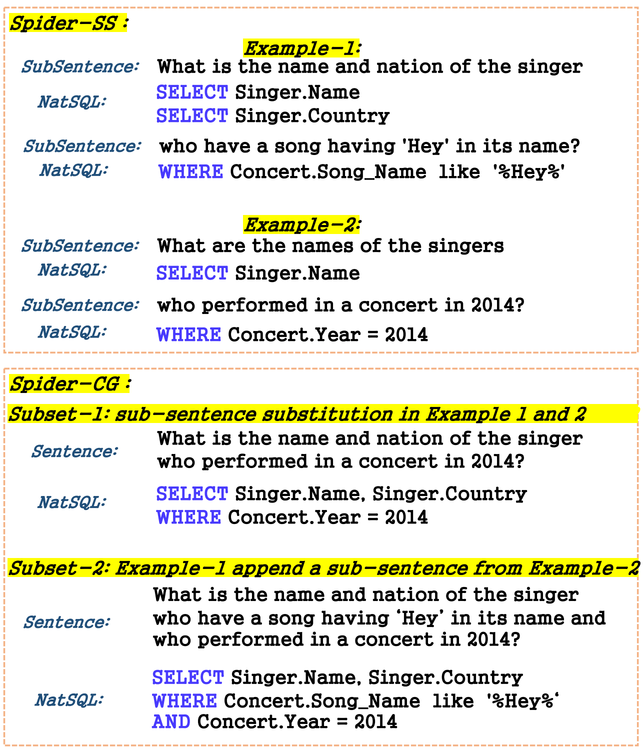
Spider−SS Example:

SubSentence:	**What type of pet**
NatSQL:	`SELECT Pets.Pettype`
SubSentence:	**is the youngest animal**
NatSQL:	`ORDER BY Pets.Pet_Age LIMIT 1`
SubSentence:	**, and how much does it weigh?**
NatSQL:	`SELECT Pets.Weight`

Figure 7.1: A natural language sentence in the original Spider benchmark is split into three sub-sentences in Spider-SS, where each sub-sentence has a corresponding NatSQL clause.

Yin et al. [2021] and Herzig and Berant [2021] propose span-based semantic parsers that predict a sub-program over an utterance span. However, these works are based on datasets where component alignment is relatively easy to achieve; but for more complex text-to-SQL, their methods cannot be used directly. For example, as shown in the lower part of Figure 7.1, to align the sub-sentence with the sub-SQL, the algorithm needs to know that '*youngest*' corresponds to '*age*', and '*weigh*' corresponds to '*weight*'. For small or single-domain settings, such an alignment algorithm can be built by establishing rules; however, there is currently no simple and feasible alignment method for large complex cross-domain text-to-SQL, as in e.g. the Spider benchmark [Yu et al., 2018b].

In this chapter, we first introduce a new dataset, Spider-SS (SS stands for *sub-sentence*), derived from Spider [Yu et al., 2018b]; Figure 7.1 compares the two. To build Spider-SS, we first design a sentence split algorithm to split every Spider sentence into several sub-sentences until indivisible. Next, we annotate every sub-sentence with its corresponding SQL clause, reducing the difficulty of this task by using the intermediate representation language NatSQL [Gan et al., 2021c], which is simpler and syntactically aligns better with natural language (NL). Spider-SS thus provides a new resource for designing models with better generalization capabilities without designing a complex alignment algorithm. Furthermore, it can also be used as a benchmark for evaluating future alignment algorithms. To our knowledge, this is the first sub-sentence-based text-to-SQL dataset.



Spider−SS :

Example−1:

SubSentence: What is the name and nation of the singer

NatSQL:
```
SELECT Singer.Name
SELECT Singer.Country
```

SubSentence: who have a song having 'Hey' in its name?

NatSQL:
```
WHERE Concert.Song_Name like '%Hey%'
```

Example−2:

SubSentence: What are the names of the singers

NatSQL:
```
SELECT Singer.Name
```

SubSentence: who performed in a concert in 2014?

NatSQL:
```
WHERE Concert.Year = 2014
```

Spider−CG :

Subset−1: sub−sentence substitution in Example 1 and 2

Sentence: What is the name and nation of the singer who performed in a concert in 2014?

NatSQL:
```
SELECT Singer.Name, Singer.Country
WHERE Concert.Year = 2014
```

Subset−2: Example−1 append a sub−sentence from Example−2

Sentence: What is the name and nation of the singer who have a song having 'Hey' in its name and who performed in a concert in 2014?

NatSQL:
```
SELECT Singer.Name, Singer.Country
WHERE Concert.Song_Name like '%Hey%'
AND Concert.Year = 2014
```

Figure 7.2: Two Spider-CG samples generated by: (1) substituting the sub-sentence with one from another example; or (2) composing sub-sentences from 2 examples in Spider-SS.

Our annotated Spider-SS provides us with sub-sentences paired with NatSQL clauses, which serve as our elements. Based on Spider-SS, we then construct a further dataset Spider-CG (CG stands for *compositional generalization*), by substituting sub-sentences with those from other samples, or composing two sub-sentences to form a more complicated sample. Spider-CG contains two subsets; Figure 7.2 shows one example for each. The first subset contains 23,569 examples generated by substituting sub-sentences. Since substitution does not increase the complexity of the NL questions, we consider most data in this subset would be close to the original, i.e., belonging to in-distribution. The second subset contains 22,030 examples generated by appending sub-sentences. Because NL questions in the second subset are longer than the original data, we consider them more complex than the original data, i.e., belonging to OOD. We demonstrate that when models are trained only on the original Spider dataset, they suffer a significant performance drop on the second OOD subset of Spider-CG, even though the domain appears in the training set.

To improve the generalization performance of text-to-SQL models, we modify several previous state-of-the-art models so that they can be applied to the Spider-SS dataset, with the model trained sub-sentence by sub-sentence. This modification obtains more than 7.8% accuracy improvement on the OOD subset of Spider-CG.

This chapter is based on [Gan et al., 2022]. Our contributions in this chapter are as follows:

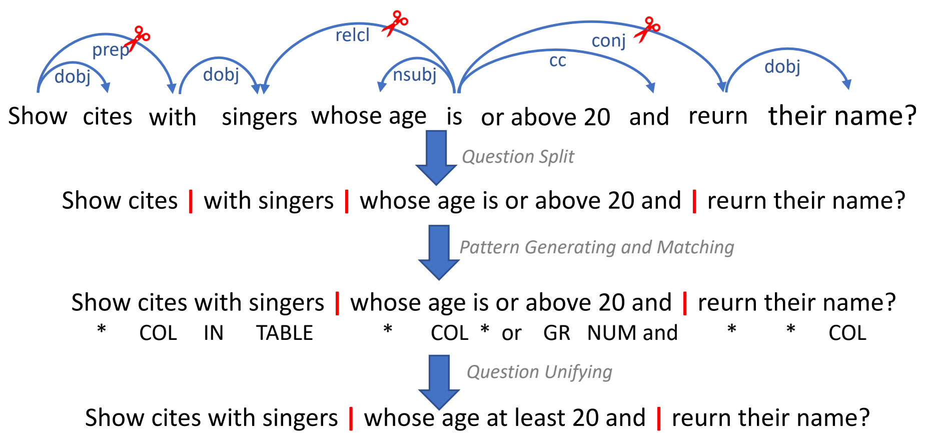
- Besides the sentence split algorithm, we introduce Spider-SS, a human-curated sub-sentence-based text-to-SQL dataset built upon the Spider benchmark, by splitting its NL questions into sub-sentences.
- We introduce the Spider-CG benchmark for measuring the compositional generalization performance of text-to-SQL models.
- We show that text-to-SQL models can be adapted to sub-sentence-based training, improving their generalization performance.

7.1 Spider-SS

7.1.1 Overview

Figure 7.1 presents a comparison between Spider and Spider-SS. Unlike Spider, which annotates a whole SQL query to an entire sentence, Spider-SS annotates the SQL clauses to sub-sentences. Spider-SS uses NatSQL [Gan et al., 2021c] instead of SQL for annotation, because it is sometimes difficult to annotate the sub-sentences with corresponding SQL clauses due to the SQL language design. The Spider-SS provides a combination algorithm that collects all NatSQL clauses and then generates the NatSQL query, where the NatSQL query can be converted into an SQL query.

The purpose of building Spider-SS is to attain clause-level text-to-SQL data avoiding the need for an alignment algorithm that is hard to build based on the complex large cross-domain text-to-SQL dataset, e.g., Spider benchmark. Besides, we can generate more complex examples through different combination of clauses from Spider-SS. Consistent with Spider, Spider-SS contains 7000 training and 1034 development examples, but Spider-SS does not contain a test set since the Spider test set is not public. There are two steps to build Spider-SS. First, design a sentence split algorithm to cut the sentence into sub-sentences, and then manually annotate the NatSQL clause corresponding to each sub-sentence.



prep
dobj
dobj
relcl
nsubj
cc
conj
dobj
Show cites with singers whose age is or above 20 and reurn their name?
Question Split
Show cites | with singers | whose age is or above 20 and | reurn their name?
Pattern Generating and Matching
Show cites with singers | whose age is or above 20 and | reurn their name?
* COL IN TABLE * COL * or GR NUM and * * COL
Question Unifying
Show cites with singers | whose age at least 20 and | reurn their name?

Figure 7.3: Dependency structure of a sentence and how to split this sentence into three sub-sentences.

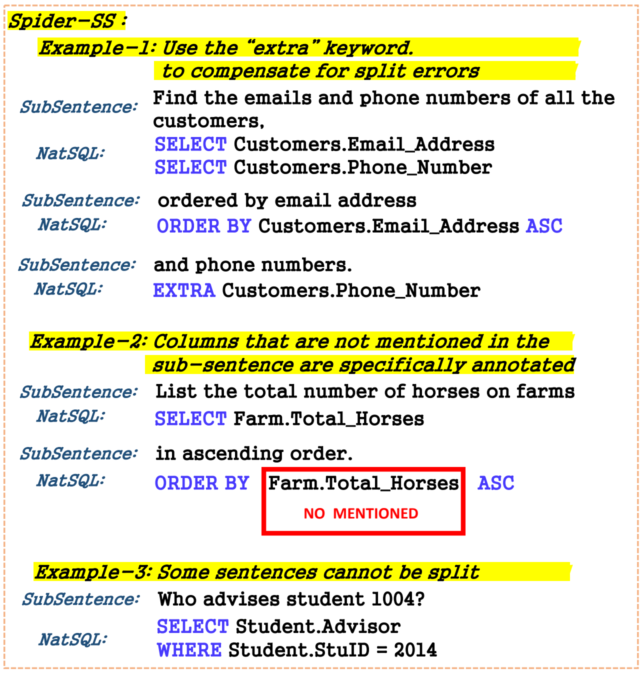
7.1.2 Sentence Split Algorithm

We build our sentence split algorithm upon the NL dependency parser spaCy [1], which provides the grammatical structure of a sentence. Basically, we split the sentence with the following dependencies: *prep, relcl, advcl, acl, nsubj, npadvmod, csubj, nsubjpass* and *conj*. According to [de Marnee and Manning, 2016], these dependencies help us separate the main clause, subordinate clauses, and modifiers. Figure 7.3 shows the dependency structure of a sentence and how to split this sentence into three sub-sentences. However, not every sentence would be split since there are some non-splittable sentences, such as the third example in Figure 7.4, with the same annotation as the Spider dataset. Although this method can separate sentences well in most cases, due to the variability of natural language, some examples cannot be perfectly split. If you are concerned about the performance of models on non-splittable sentences, you can add the original Spider data for training.

To address the remaining issues in sentence split, we design some refinement steps tailored to text-to-SQL applications. For example, when the phase of a schema column or table is accidentally divided into two sub-sentences, these two sub-sentences are automatically concatenated. Besides, when there is only one word in a sub-sentence, the corresponding split should also be undone.

We sampled 500 examples from the Spider-SS development set to evaluate the acceptability of splitting results manually, and only $< 3\%$ of the splitting results are unsatisfactory. For example, in the splitting results of the first example in Figure 7.4, the last two sub-sentences should be combined to correspond to "**ORDER BY** Customer.Email_Address, Customer.Phone_Number

[1]https://github.com/explosion/spaCy


Spider−SS :

Example−1: Use the "extra" keyword. to compensate for split errors

SubSentence: Find the emails and phone numbers of all the customers.

NatSQL:
```
SELECT Customers.Email_Address
SELECT Customers.Phone_Number
```

SubSentence: ordered by email address

NatSQL:
```
ORDER BY Customers.Email_Address ASC
```

SubSentence: and phone numbers.

NatSQL:
```
EXTRA Customers.Phone_Number
```

Example−2: Columns that are not mentioned in the sub−sentence are specifically annotated

SubSentence: List the total number of horses on farms

NatSQL:
```
SELECT Farm.Total_Horses
```

SubSentence: in ascending order.

NatSQL:
```
ORDER BY Farm.Total_Horses ASC
```

Example−3: Some sentences cannot be split

SubSentence: Who advises student 1004?

NatSQL:
```
SELECT Student.Advisor
WHERE Student.StuID = 2014
```

Figure 7.4: Spider-SS examples in three special cases.

ASC ". In this example, we did not simply give an "**ORDER BY** Customer.Phone_Number **ASC** " to the last sub-sentence, because it does not mention anything related to "**ORDER BY** ". Here, we introduce "*extra*", a new NatSQL keyword designed for the Spider-SS dataset, indicating that this sub-sentence mentions a column that temporarily does not fit in any other NatSQL clauses. When combining NatSQL clauses into the final NatSQL query, the combining algorithm determines the final position for the "*extra*" column based on the clauses before and after. Note that even if there is a small proportion of unsatisfactory splitting results, as long as the model trained on Spider-SS can give the correct output according to the input sub-sentence, the quality of the sub-sentences itself does not strongly affect the model utility.

7.1.3 Data Annotation

When we get the split results from the last step, we can start data annotation. We give precise annotations based on the sub-sentence content, such as the "*extra*" column annotation discussed in the last subsection. Besides, if the description of the schema column is missing in the sub-sentence, we will give the schema column an additional "*NO MENTIONED*" mark. For example, in the second example of Figure 7.4, the "*in ascending order*" sub-sentence does not mention the "*Farm.Total_Horses*" column. Therefore, we add a "*NO MEN-*

TIONED" mark for it. For those sub-sentences that do not mention anything related to the query, we give a "*NONE*" mark, representing there are no NatSQL clauses.

Since the annotation is carried out according to the sub-sentence content, the equivalent SQL that is more consistent with the sub-sentence will be preferred to the original SQL. Similarly, if the original SQL annotation is wrong, we correct it according to the content.

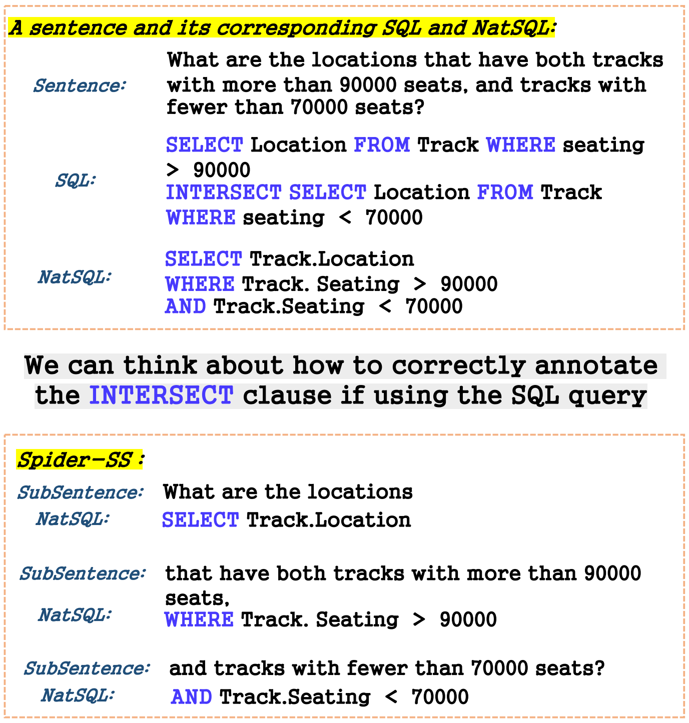
We annotate the sub-sentence using NatSQL instead of SQL, where NatSQL is an intermediate representation of SQL, only keeping the *SELECT, WHERE, and ORDER BY* clauses from SQL. Since some sub-sentences need to be annotated with *GROUP BY* clause, we choose the version of NatSQL augmented with *GROUP BY*. We did not use SQL directly because it is difficult to annotate in some cases, such as the SQL example in Figure 7.5. The difficulty is that there are two *SELECT* clauses in this SQL query, but none of the sub-sentences seem to correspond to two *SELECT* clauses. In addition, considering that the two *WHERE* conditions correspond to different *SELECT* clauses, the annotation work based on SQL is far more difficult to complete. As shown in Figure 7.5, we can use NatSQL to complete the annotation quickly, while the NatSQL can be converted back to the target SQL.

We build an annotation tool to show the sub-sentence and sub-SQL split from a question-NatSQL pair. During annotation, the annotators select the corresponding sub-SQL for sub-sentences. In rare cases, if there is no suitable sub-SQL, the annotators would write a new one, such as the example-1 in Figure7.4. We recruit two graduate students major in computer science to annotate the dataset manually. They are trained with a detailed annotation guideline and some samples. One is allowed to start after his trial samples are approved by the whole team. Each example is annotated twice. If the annotations are different, the final annotation will be decided by a discussion. If two annotators discuss and conclude that one of the annotations is wrong and the other is correct, the correct annotation is retained. Otherwise, the authors will annotate this example if no such conclusion can be drawn.

7.2 Spider-CG

7.2.1 Overview

Spider-CG is a synthetic dataset, which is generated by recombining the sub-sentences of Spider-SS. There are two recombination methods. The first is sub-sentence substitution between different examples, and the other is to append a sub-sentence into another sentence. To facilitate the follow-up discussion, we



A sentence and its corresponding SQL and NatSQL:

Sentence: What are the locations that have both tracks with more than 90000 seats, and tracks with fewer than 70000 seats?

SQL:
```
SELECT Location FROM Track WHERE seating > 90000
INTERSECT SELECT Location FROM Track WHERE seating < 70000
```

NatSQL:
```
SELECT Track.Location
WHERE Track. Seating > 90000
AND Track.Seating < 70000
```

We can think about how to correctly annotate the INTERSECT clause if using the SQL query

Spider−SS :

SubSentence: What are the locations
NatSQL: `SELECT Track.Location`

SubSentence: that have both tracks with more than 90000 seats,
NatSQL: `WHERE Track. Seating > 90000`

SubSentence: and tracks with fewer than 70000 seats?
NatSQL: `AND Track.Seating < 70000`

Figure 7.5: It is difficult to annotate if using the SQL instead of NatSQL.

named the Spider-CG subset generated by the sub-sentence substitution method **CG-SUB**, and the other named **CG-APP** where CG denotes Spider-CG, SUB stands for substitution, and APP represents append.

In CG-SUB, there are 20,686 examples generated from the Spider-SS training set, while 2,883 examples are generated from the development set. In CG-APP, examples generated from training and development sets are 18,793 and 3,237, respectively. Therefore, the Spider-CG contains 45,599 examples, around six times the Spider dataset. We can further append sub-sentences to the CG-SUB examples if more data is needed.

7.2.2 Generation Algorithm

According to Algorithm 2, we can generate the CG-SUB and CG-APP based on compositional elements. Each element contains one or more sub-sentences with corresponding NatSQL clauses from Spider-SS, where these NatSQL can only be *WHERE or ORDER BY* clauses. Thus, Algorithm 2 only substitute and append the *WHERE and ORDER BY* clauses, and does not modify the *SELECT* clause. We collect the sub-sentences for compositional elements by scanning all sub-sentence from start to end or from end to start and stopping

Algorithm 2 Generate CG-SUB and CG-APP dataset in a certain domain

```
Input: $e\_list$                          ▷ All compositional elements in a domain
Output: $cg\_sub$ and $cg\_app$           ▷ CG-SUB and CG-APP dataset in a certain domain
1: for Every $element_1$ in $e\_list$ do
2:     for Every $element_2$ in $e\_list$ do
3:         if $element_1$ != $element_2$ then
4:             if $element_1$.can_be_substituted_by( $element_2$ ) then
5:                 $cg\_sub$.append( $element_1$.generate_substitution_example( $element_2$ ) )
6:             if $element_1$.can_append( $element_2$ ) then
7:                 $cg\_app$.append( $element_1$.generate_appending_example( $element_2$ ) )
8: return $cg\_sub$, $cg\_app$
```

Ques	Show the name of employees named Mark Young ?
SQL	**SELECT** name **FROM** employee **WHERE** name = 'Mark Young'

Table 7.1: One acceptable but not perfect examples in the Spider-CG.

when encountering clauses except *WHERE and ORDER BY*. For example, we generate a compositional element containing the last two sub-sentences of the Spider-SS example in Figure 7.5. In contrast, no element is extracted from the example in Figure 7.1. It should be noted that elements in a domain cannot be used in another because the schema items are different. So as many domains as there are, it needs to run Algorithm 2 as many times.

To ensure that the generated Spider-CG sentence contains the required information, the compositional element needs to contain all the information needed to derive the target NatSQL clause. Thus some sub-sentence can not be a compositional element, such as the last sub-sentence of examples 1 and 2 in Figure 7.4. Among them, example 1 misses *ORDER BY* information; example 2 misses *Total_Horses* column information. In contrast, the sub-sentence of the two Spider-SS examples in Figure 7.2 contains the required information and can be compositional elements. So, we can filter out the sub-sentences containing the "*NO MENTIONED*" and "*extra*" label, and collect the rest as compositional elements.

The '*can_be_substituted_by*' and '*can_append*' function in Algorithm 2 are used to ensure that the generated sentences are reasonable. For the convenience of discussion, we refer to them as '*sub*' and '*app*' functions for short. These two functions examine the generated sentences from complexity, logic and coherence.

Complexity checks are used to limit the complexity of the generated examples to no more complex than the upper bound of the Spider dataset. On the NatSQL side, both functions do not allow the generated NatSQL containing: 1) more than one subqueries; 2) more than one *HAVING* condition; 3) more than three *WHERE* conditions; 4) more than one *ORDER BY* clause; 5) new conditions for a subquery. On the NL side, since the substitution did not clearly increase the sentence complexity, only the '*app*' function performs the NL complexity checks to restrict the number of sub-sentence to less than 4.

Logic checks are used to prevent generating contradictory examples. First, logic checks filter out examples with repeated *WHERE* conditions. Then, it filters out examples whose *WHERE* condition negates the query content, e.g., *what is name of student that do not have any student.* Finally, since the *GROUP BY* clause is often expressed implicitly, substituting or appending elements containing the *GROUP BY* clause may introduce logical errors. Thus, logic checks require the *GROUP BY* clauses to be the same if they exist.

Coherence checks are used to ensure that the expression of the generated sentence is coherent. As discussed in Section 7.1.2, we separate a sentence into main clause, subordinate clauses, and modifiers. The main clause expresses what you want to query, i.e., corresponding to the SELECT clause. Subordinate clauses and modifiers are restrictions on the query, i.e., corresponding to *WHERE* and *ORDER BY* clauses. Therefore, compositional elements only contain subordinate clauses and modifiers. The way to ensure the coherence of sentences by *sub* function is to require the substitution sub-sentences modify the same noun. Suppose the schema table of the NatSQL in a compositional element appears in advance. In that case, we consider its sub-sentence modifies the table noun because repeating a known object [2] can only be a further modification. However, if the schema table has not appeared before, we consider that the sub-sentence modifies its previous word since a subordinate clause usually comes immediately after the noun it describes.

There is a high similarity between the *app* and *sub* function, but the inspection between the substituted elements is changed to the inspection between the new element and the last element in the original sentence. Therefore, the appended sub-sentence must modify the same noun as the last sub-sentence. If a compositional element passes the *app* function, we use the word '*and*' or '*or*' to connect it where the word '*or*' can only connect a *WHERE* condition. Table 7.2 discuss some examples for ease of understanding.

[2]A table is usually an object whose attributes are its columns in relational databases.

Spider sentence: Show name for all singers ordered by age from the oldest to the youngest. How many concerts are there in year 2014 or 2015? Generate new sentence by appending: Show name for all singers ordered by age from the oldest to the youngest and in year 2014 or 2015? Coherence checks: Failed to pass the coherence checks due to the modified noun of the two sub-sentences being different. In the same way, the '*Show name for all singers in year 2014 or 2015?*' can not pass.
Spider sentence: Show name for all singers ordered by age from the oldest to the youngest. What is the nation of the singer who have a song having ' Hey ' in its name? Generate new sentence by appending: What is ... who have a song having ' Hey ' in its name and ordered by age from the oldest to the youngest. Coherence checks: Pass the coherence checks. In the same way, the '*what is ... singer ordered by age from the oldest to the youngest .*' also pass.
Spider sentence: What are the titles of the books whose writer is not 'Elaine Lee'? List the writers who have written more than one book. Generate new sentence by appending: What are the titles of the books whose writer is not 'Elaine Lee' and who have written more than one book. Coherence checks: Failed to pass the coherence checks due to the modified noun of the two sub-sentences being different. In the same way, the '*What are the titles of the books who have written more than one book.?*' can not pass.
Spider sentence: List the writers who have written more than one book. Show writers who have published a book with price more than 40. Generate new sentence by appending and substituting: List the writers who have written more than one book and who have published a book with price more than 40. List the writers who have written more than one book or who have published a book with price more than 40 . Show writers who have published a book with price more than 40 and who have written more than one book . Show writers who have published a book with price more than 40 or who have written more than one book. List the writers who have written more than one book. Show writers who have written more than one book. Coherence checks: All these sentence pass the coherence checks.

Table 7.2: Some examples of successful or unsuccessful passing the coherence checks.

7.2.3 Quality Evaluation

We consider that the quality of a text-to-SQL sentence is determined by two criteria: containing the required information and being reasonable. The 'information' criterion requires a sentence that contains all the information needed to derive the target NatSQL. The 'reasonable' criterion requires a sentence that is logically correct and whose representation is fluent and easy to understand. We randomly sampled 2000 examples from the Spider-CG dataset, around 99% of which are acceptable, i.e., they meet the two criteria. The evaluation is conducted manually by a computer science graduate with good knowledge of text-to-SQL. However, these acceptable examples do not mean that there are no grammatical errors and they may be meaningless. We give one acceptable but not perfect examples in Table 7.1, where the sentence is meaningless because the content it wants to query is the condition it gave. Besides, there are around 5% NatSQL queries in these acceptable examples that can not be converted to the correct SQL. This problem can be solved by a well-designed

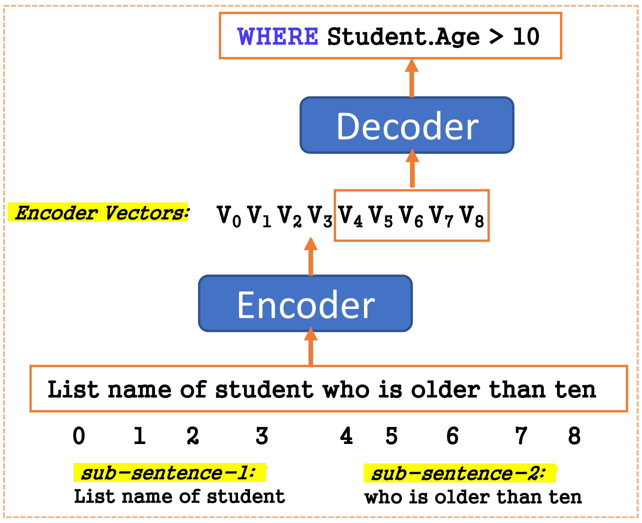


Figure 7.6: A example of encoding the whole sentence but decoding only the sub-sentence.

database schema or updating the NatSQL conversion function in the future.

7.3 Model

Existing text-to-SQL models input a sentence and output the corresponding SQL query. So the easiest way to think of using the Spider-SS dataset is to train the model where inputting sub-sentence and outputting the corresponding NatSQL clauses. However, this method is not workable because it will lose some essential schema information. For example, if you only look at the third sub-sentence in Figure 7.1, you do not know whether it enquires about the weight of pets or people.

In order to take into account the context and the sub-sentence data of Spider-SS, we propose that a seq2seq model can encode the whole sentence but decode only the sub-sentence. Figure 7.6 presents the workflow of encoding the whole sentence but only decoding the sub-sentence of '*who is older than ten*' and outputting the corresponding NatSQL clause. Based on this modification, a seq2seq text-to-SQL model can be adapted to the Spider-SS. Although previous span-based semantic parsers [Yin et al., 2021, Herzig and Berant, 2021] can work with aligned annotations based on the Spider-SS dataset, none of them are designed for complex text-to-SQL problems. Our modification idea is similar in principle to the span-based semantic parsers, but we did not change the existing model according to the span-based because our modification idea has a smaller workload.

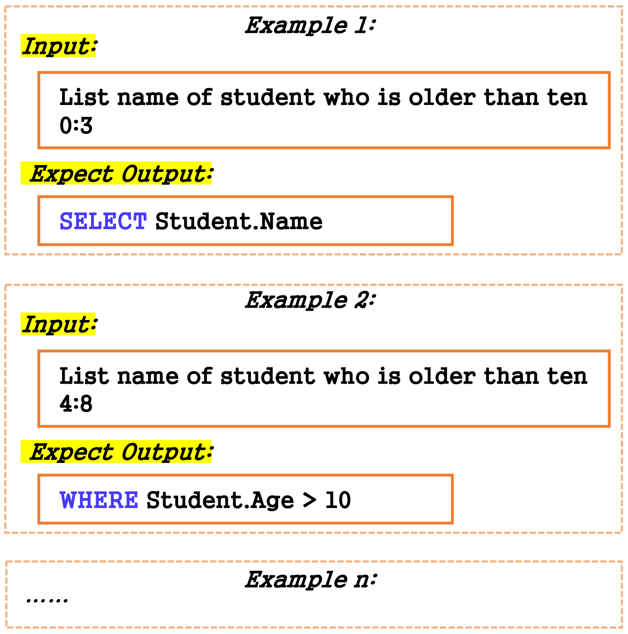


Figure 7.7: A Spider-SS example is split into two examples for training and evaluation.

In general, we can make the seq2seq-based text-to-SQL models adapt to the Spider-SS in three steps. (1) Data preprocess. Split the Spider-SS examples by sub-sentence. For example, the example in Figure 7.6 is split to two examples shown in Figure 7.7. (2) Model modification. After data preprocessing, there are two input data for a model. The first input is an entire question that directly goes to the encoder. The second input is the sub-sentence indexes, which are used to select the encoder output, as shown in Figure 7.6. (3) Output combination. Since the model output may be only a clause, not a complete NatSQL query, we generate the final NatSQL query after the model outputting all NatSQL clauses.

7.4 Experiment

7.4.1 Experimental Setup

Dataset. We evaluate the previous state-of-the-art models on the Spider-CG and Spider [Yu et al., 2018b] datasets. Since the Spider test set is not publicly accessible, Spider-CG does not contain a test set. As discussed in Section 7.2.1, we divide the Spider-CG into two subsets: CG-SUB and CG-APP. Therefore, there are five evaluation sets:

- **Spider$_\text{D}$**: the original Spider development set with 1,034 examples for *cross-domain in-distribution* text-to-SQL evaluation.
- **CG-SUB$_\text{T}$**: the CG-SUB training set, containing 20,686 examples generated from Spider-SS training set by substituting sub-sentences. CG-SUB$_\text{T}$ can be used for *in-domain in-distribution* text-to-SQL evaluation.
- **CG-SUB$_\text{D}$**: the CG-SUB development set containing 2,883 examples for *cross-domain in-distribution* text-to-SQL evaluation.
- **CG-APP$_\text{T}$**: the CG-APP training set, containing 18,793 examples generated from Spider-SS training set by appending sub-sentences. CG-APP$_\text{T}$ can be used for *in-domain out-of-distribution* [3] text-to-SQL evaluation.
- **CG-APP$_\text{D}$**: the CG-APP development set containing 3,237 examples for *cross-domain out-of-distribution* text-to-SQL evaluation.

Our evaluation is based on the exact match metric defined in the original Spider benchmark. The exact match metric measures whether the syntax tree of the predicted query without condition values is the same as that of the gold query. All models are only trained on 7000 Spider or Spider-SS training examples.

Models. We evaluate the following open-source models that reach competitive performance on Spider:

- **GNN**: The GNN [Bogin et al., 2019a] model using the GLOVE [Pennington et al., 2014] embeddings.
- **RATSQL**: The RATSQL [Wang et al., 2020] model using the GLOVE embeddings.
- **RATSQL$_\text{B}$**: The RATSQL model using the BERT [Devlin et al., 2019] embeddings.
- **RATSQL$_\text{G}$**: The RATSQL model using the GAP [Shi et al., 2021] embeddings.
- $_\text{(N)}$: This subscript indicates that the model use NatSQL instead of SQL.
- $_\text{(S)}$: This subscript indicates that the model is modified according to Section 7.3 and trained on Spider-SS. Besides, since Spider-SS is annotated by NatSQL, this subscript also indicates that the model uses NatSQL instead of SQL.

[3]Out-of-distribution means that the difficulty distribution is different from the Spider; see Table 7.4. Section 7.2.2 discusses the removal of overly complex examples to ensure that Spider-CG's SQL does not exceed the complexity upper bound of the Spider.

Dataset	**Exact Match**	**Execution Match**
Training Set	90.7%	93.3%
Development Set	94.8%	95.2%

Table 7.3: Use exact match and execution match metrics to evaluate the difference between the SQL in Spider and the SQL generated by NatSQL in Spider-SS.

Dataset	**easy**	**medium**	**hard**	**extra**
Spider$_{\mathrm{D}}$	24.1%	43.1%	16.8%	16.1%
CG-SUB$_{\mathrm{T}}$	28.6%	38.0%	21.1%	12.3%
CG-SUB$_{\mathrm{D}}$	37.6%	38.4%	12.0%	12.0%
CG-APP$_{\mathrm{T}}$	3.3%	31.4%	26.0%	39.3%
CG-APP$_{\mathrm{D}}$	2.4%	44.3%	22.9%	30.4%

Table 7.4: The difficulty distribution of five different evaluation sets.

Implementations. All experiments were performed on a machine with an Intel i5 9600 3.1GHz processor and a 24GB RTX3090 GPU. All models keep their original hyperparameters except the RATSQL$_{\mathrm{B(S)}}$. RATSQL$_{\mathrm{B(S)}}$ cannot converge on the original parameters until we reduce the learning rate of model from 7.444e-04 to 1e-04 and raise the learning rate of BERT from 3e-06 to 1e-05. We did not conduct a hyperparameter search, so the model trained on Spider-SS may improve performance through other parameters.

7.4.2 Dataset Analysis

Spider-SS. Table 7.3 presents the difference between the SQL in Spider and the SQL generated by NatSQL in Spider-SS. Our evaluation results are lower than the original NatSQL dataset [Gan et al., 2021c] because the Spider-SS uses equivalent SQL and corrects some errors, as discussed in Section 7.1.3. Some equivalent and corrected SQL cannot get positive results in exact match metric and execution match. Therefore, the model trained on Spider-SS may not be ideal for chasing the Spider benchmark, especially based on the exact match metric. Similarly, the RATSQL$_{\mathrm{G}}$ extending NatSQL had achieved a previous SOTA result in the execution match of the Spider test set but get a worse result than the original in the exact match [Gan et al., 2021c]. Thus, we recommend using NatSQL-based datasets to evaluate models trained on NatSQL.

Spider-CG. Table 7.4 presents the difficulty distribution of five different evaluation sets. The difficulty criteria are defined by Spider benchmark, including *easy, medium, hard* and *extra hard*. Experiments show that the more difficult the SQL is, the more difficult it is to predict correctly [Wang et al., 2020, Shi

Dataset	Deviation <= 1	Deviation <= 2
CG-SUB$_{\text{T}}$	93.2%	94.4%
CG-SUB$_{\text{D}}$	92.9%	94.1%
CG-APP$_{\text{T}}$	86.0%	90.4%
CG-APP$_{\text{D}}$	88.9%	92.6%

Table 7.5: The similarity between sub-sentences in Spider-SS and Spider-CG generated by the same split algorithm under the deviation of one or two tokens.

et al., 2021, Gan et al., 2021c]. It can be found from Table 7.4 that the difficulty distribution of CG-SUB$_{\text{T}}$ and CG-SUB$_{\text{D}}$ is similar to that of Spider$_{\text{D}}$. The similar distributions among CG-SUB$_{\text{T}}$, CG-SUB$_{\text{D}}$, and Spider$_{\text{D}}$ support our view that the examples generated by the substitution method are in-distribution.

On the other hand, the difficulty distributions of CG-APP$_{\text{T}}$ and CG-APP$_{\text{D}}$ are obviously different from that of Spider$_{\text{D}}$. Due to appending the sub-sentence, the NL and SQL in CG-APP become more complex, where the proportion of SQL in *extra hard* increased significantly, while *easy* was the opposite.

7.4.3 Sentence Split Algorithm Evaluation

We generate the Spider-CG based on the combination of Spider-SS sub-sentences split by the algorithm introduced in Section 7.1.2. We can reuse this algorithm to split the sentence in Spider-CG and then compare the splitting results with the Spider-SS sub-sentences to evaluate the stability of the splitting algorithm. We consider that a deviation of one or two tokens in the splitting result is acceptable. For example, in Figure 7.1, we consider that putting the comma of the third sub-sentence into the second sub-sentence does not change the meaning of sub-sentences, same for moving both the comma and the word 'and'.

Table 7.5 presents the similarity between sub-sentences in Spider-SS and Spider-CG, which are generated by the same split algorithm under the deviation of one or two words. The similarity exceeds 90% in all evaluation set when two deviation words are allowed. Considering that the model trained on the Spider-SS does not require consistent split results, as discussed in Section 7.1.2, the similarity results of the splitting algorithm are good enough. The similarity of CG-SUB is higher than that of CG-APP, which means the more complex the sentence, the greater the challenge to the algorithm. Although the algorithm has been refined on the training set, the similarity between training and development in CG-SUB and CG-APP is close, showing that the algorithm performs consistently for sentences in unseen domains. In summary, we consider that as long as the sentences are not more complex than CG-APP, the algorithm can be used stably in other text-to-SQL datasets.

Approach	Spider_D	CG-SUB_T	CG-SUB_D	CG-APP_T	CG-APP_D
$\textbf{RATSQL}_G$	72.7%	80.9%	70.3%	45.2%	44.2%
$\textbf{RATSQL}_{G(N)}$	73.9%	90.2%	75.0%	67.8%	60.5%
$\textbf{RATSQL}_{G(S)}$	**74.5%**	**91.4%**	**76.7%**	**82.5%**	**68.3%**
$\textbf{RATSQL}_B$	72.0%	79.5%	72.0%	45.1%	47.2%
$\textbf{RATSQL}_{B(N)}$	**72.1%**	83.2%	69.4%	54.6%	53.1%
$\textbf{RATSQL}_{B(S)}$	71.9%	**91.0%**	**72.6%**	**79.8%**	**61.5%**
$\textbf{RATSQL}_{(N)}$	63.2%	79.1%	60.7%	40.6%	34.5%
$\textbf{RATSQL}_{(S)}$	**64.7%**	**88.8%**	**63.3%**	**72.1%**	**44.1%**
$\textbf{GNN}_{(N)}$	**54.4%**	67.3%	**57.5%**	30.4%	25.1%
$\textbf{GNN}_{(S)}$	49.3%	**71.9%**	51.8%	**52.1%**	**34.6%**

Table 7.6: Exact match accuracy on evaluation sets.

7.4.4 Model Results

Table 7.6 presents the exact match accuracy on the five different evaluation sets. In the two OOD datasets, CG-APP_T and CG-APP_D, the performance of all models has dropped by about 10% to 30%. However, the models trained on Spider-SS significantly outperform those trained on Spider when evaluated on the OOD datasets. We use the sentence split algorithm to split every sentence before inputting the models with subscript (S). Although the split sub-sentences are not completely consistent with those seen during training, it did not prevent the models with subscript (S) from getting good performance, i.e., the $\text{RATSQL}_{G(S)}$ consistently outperforms all other models on all evaluation sets. These results demonstrate that the sub-sentence-based method can improve the generalization performance. The limitation is that the method may not be compatible with the original model, e.g., original hyperparameters in $\text{RATSQL}_{B(S)}$ are not workable, and the performance of GNN on the Spider_D and CG-SUB_D is degraded.

Each model has a close result between the unseen Spider_D and CG-SUB_D, indicating that from the perspective of the model, the synthetic sentences are pretty similar to NL. Therefore, we believe the performance on CG-SUB_D can be generalized to the real world. Moreover, considering that the algorithms for generating CG-SUB_D and CG-APP_D are close (see Section 7.2.2), we can further speculate that the synthetic sentences of CG-APP_D are also close to natural language.

The models with NatSQL is significantly better than that without NatSQL when evaluated on Spider-CG. One of the reasons is that the training data of Spider and Spider-SS are about 10% different, which leads to the performance degradation in the model trained on Spider when evaluated on the SQL generated by the NatSQL of Spider-SS, and vice versa. On the other hand, experiments in [Gan et al., 2021c] show that NatSQL improve the model performance in *extra hard* SQL. Therefore, $\text{RATSQL}_{\text{G(N)}}$ and $\text{RATSQL}_{\text{B(N)}}$ suffer less performance degradation in CG-APP_{T} and CG-APP_{D} than RATSQL_{G} and RATSQL_{B}.

7.5 Summary

We introduce Spider-SS and Spider-CG for measuring compositional generalization of text-to-SQL models. Specifically, Spider-SS is a human-curated sub-sentence-based text-to-SQL dataset built upon the Spider benchmark. Spider-CG is a synthetic text-to-SQL dataset constructed by substituting and appending sub-sentences of different samples, so that the training and test sets consist of different compositions of sub-sentences. We found that the performance of previous text-to-SQL models drop dramatically on the Spider-CG OOD subset, while modifying the models to fit the segmented data of Spider-SS improves compositional generalization performance.

Chapter 8

Conclusions

In this thesis, we studied text-to-SQL from two perspectives; performance and robustness. To this end, we improved model performance by introducing the NatSQL that bridges the mismatch between natural language (NL) and SQL. Additionally, we found that the robustness of the model is poor when breaking the schema linking or requiring generalization to more complex compositional examples.

The intermediate representation (NatSQL) was used almost exclusively throughout the whole thesis. In Chapter 4, we introduce NatSQL, a new SQL IR that reduces the difficulty of schema linking and simplifies the SQL structure. NatSQL improves the performance of several baseline models and helps them generate the executable SQL. In Chapter 5, experiments on NatSQL show that the interpretable schema linking module works better with NatSQL because NatSQL removes many implicit schema items. In Chapter 6, we use NatSQL to build the sub-sentence-based Spider-SS dataset since it is difficult to annotate using SQL.

For robustness, Chapter 5 studies the model robustness against synonym substitution on schema item words. We tested several text-to-SQL models, and they failed to handle the synonym substitution. We propose MAS and adversarial training methods to improve model robustness against the synonym substitution. The synonym substitution can be considered to have broken the schema linking. Thus, Chapter 6 studied the text-to-SQL schema linking and found that previous models rely on the EMSL. We argue that the EMSL is not the icing on the cake, but it is the one that introduces the vulnerability, and it can be replaced by better input encoding. In Chapter 6, we study the model's robustness against compositional generalization. Again, we found the existing models unable to generate correct SQL for the more complex compositional question, even though all sub-sentences have been seen during training.

We modified these models to learn the sub-sentences, not the whole question, which improved their compositional generalization ability. We highlight that compositional generalization in text-to-SQL requires compositional parsers.

8.1 Future Directions

We conclude this thesis with a brief discussion of future directions for research on text-to-SQL.

Text-to-SQL Benchmarks. The WikiSQL and Spider are large-scale cross-domain text-to-SQL benchmarks and have been widely used. The main difference between them is that the schema structure and SQL are more complex in the Spider. However, we can further complicate the Spider examples to construct a more challenging benchmark. For example, we can add examples requiring more complex reasoning, such as basic mathematical operations. Additionally, the new benchmark can further improve the evaluation metrics. The Spider benchmark provides two metrics for evaluation. The first metric is exact match that measures whether the predicted query without both condition values and *JOIN ON* clause as a whole is equivalent to the gold query. However, *JOIN ON* is essential when there are several possible *JOIN ON* paths. Besides, the Spider exact match metric cannot correctly evaluate the sub-query and equivalent SQL. Thus, Zhong et al. [2020a] further studies the equivalent SQL evaluation based on the original Spider exact match metric, but there is still room for improvement.

The second metric is execution match, which measures whether the execution result of the predicted query from the database is the same as the gold query. Currently, this metric simply checks whether the returned data is strictly consistent, but this design is not thoughtful enough. For example, if the user queries data but does not require its returned order, the different order results would be negative. In addition, the current metric does not allow returning more column data than the users expect; if, for example, the user wants the student's age, the metric would give a negative to the return data containing the student's name and age. In general, a more complex cross-domain text-to-SQL dataset with a more reasonable metric is worth researching.

Intermediate Representation. NatSQL cannot cover all SQL and is designed based on the Spider benchmark. With the further complexity of SQL and the database structure, the current NatSQL seems insufficient. For example, the NatSQL does not contain a *JOIN ON* clause. If the *JOIN ON* clause is to become more complex than the examples in the Spider, it will be necessary to design a more powerful IR. On the other hand, a new database paradigm can be proposed from the perspective of Natural Language Interface to Database (NLIDB) to restrict the complexity of the schema database. For example, the new paradigm can require that columns with the same meaning be labeled in the same group when creating a database, which makes it easier for IR to generate *JOIN ON* clause.

At present, we have completed the conversion from NatSQL to SQL, but not from SQL to NatSQL. SQL to NatSQL is a lot more complicated than the reverse. For example, the two NatSQL queries in Table 8.1 can be converted to the same SQL, which means that it is difficult to determine which NatSQL is the target query for SQL to NatSQL conversion. The SQL to NatSQL conversion requires NL analysis. It can be inferred from the NL content of 'the most concerts' that we should select the first NatSQL query containing the '**ORDER BY** count(concert.*) **DESC LIMIT** 1' clause. Therefore, it is an interesting future direction for studying the SQL to NatSQL conversion.

NL:	What is the name and capacity of the stadium with the most concerts after 2013?
SQL:	**SELECT** T2.name , T2.capacity **FROM** concert AS T1 **JOIN** stadium AS T2 **ON** T1.stadium_id = T2.stadium_id **WHERE** T1.year >= 2014 **GROUP BY** T2.stadium_id **ORDER BY** count(*) **DESC LIMIT** 1
NatSQL:	**SELECT** stadium.name , stadium.capacity **WHERE** T1.year >= 2014 **ORDER BY** count(concert.*) **DESC LIMIT** 1
NatSQL:	**SELECT** stadium.name , stadium.capacity **WHERE** T1.year >= 2014 **ORDER BY** count(stadium.*) **DESC LIMIT** 1

Table 8.1: The two NatSQL queries can be converted to the same SQL query.

Text-to-SQL Model. Most existing text-to-SQL models do not care about the correctness in the *JOIN ON* clause because the Spider exact match metric does not check it. As discussed above, it is important to generate the correct *JOIN ON* clause. The difficulty in generating a correct *JOIN ON* clause is that most *JOIN ON* clauses are implicitly expressed in the question. A model with *JOIN ON* clause generation and reasoning ability is important for complex text-to-SQL problems.

Conversely, the phrase-based (sub-sentence-based) text-to-SQL model is also worth studying. Our experiments on Spider-CG have shown that models adapting to the Spider-SS can improve their compositional generalization ability. However, the original model is not designed based on sub-sentences. We believe the community would benefit from a newly designed phrase-based text-to-SQL model. We have submitted a paper related to this direction.

Understanding Domain knowledge. Gan et al. [2021b] studied the domain knowledge in cross-domain text-to-SQL, and they found that existing models do not understand domain knowledge. However, this work does not provide any solution to help the model understand the domain knowledge. We believe understanding the domain knowledge is essential for models to generalize to unseen domains.

Handling Large-scale Database. The database in current benchmarks is usually small-scale, containing less than twenty columns and rows. The small-scale database makes schema linking relatively easy and meets the input length limitation of many pre-trained language models, such as BERT. Besides, fewer data rows make it easy to scan the entire database to establish cell value linking. However, in the real-world scenario, the database usually consists of thousands of rows and columns, which raises a new challenge to existing neural text-to-SQL models. In particular, the challenges include: (1) for a long sequence of table schemas, how to encode it and how to construct a proper schema linking? (2) for tons of data, how to efficiently construct a cell value linking without scanning the entire database?

8.2 Software and Data

We release the following datasets and code related to this thesis:

- NatSQL: https://github.com/ygan/NatSQL. We released the NatSQL query for Spider benchmarks and the NatSQL to SQL conversion code;
- RATSQL+NatSQL model (available upon request). This code was modified from the original RATSQL and can be used for studying how to modify the existing models adapting to the NatSQL;
- Spider-Syn: https://github.com/ygan/Spider-Syn. We released the Spider-Syn dataset for evaluating the synonym substitution. We also released the code for automatically generating the synonym substitution examples;
- Analysis of schema linking: It includes code of RATSQL$_O$ and Spider-T1 to SPider-T3 datasets (the code and datasets are yet not accessible to facilitate double-blind reviewing);
- Spider-SS: https://github.com/ygan/SpiderSS-SpiderCG. We released the Spider-SS dataset for studying sub-sentence-based text-to-SQL;

- Spider-CG: https://github.com/ygan/SpiderSS-SpiderCG. We provided the preprocessed Spider-CG for ease of use. Alternately, you could generate it from Spider-SS based on our released code.

Bibliography

Amal Menzli. Graph neural network and some of gnn applications: Everything you need to know, 2021. URL `https://neptune.ai/blog/graph-neural-network-and-some-of-gnn-applications`. [Online; accessed 12-March-2022].

Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. Towards a Theory of Natural Language Interfaces to Databases. In *Proceedings of the 8th International Conference on Intelligent User Interfaces*, pages 149–157, 2003. URL `http://doi.acm.org/10.1145/604045.604070`.

Jacob Andreas, Andreas Vlachos, and Stephen Clark. Semantic parsing as machine translation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 47–52, Sofia, Bulgaria, August 2013. Association for Computational Linguistics. URL `https://aclanthology.org/P13-2009`.

I Androutsopoulos, G D Ritchie, and P Thanisch. Natural language interfaces to databases – an introduction. *Natural Language Engineering*, 1(1):29–81, 1995. doi: 10.1017/S135132490000005X.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on Freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544, Seattle, Washington, USA, October 2013. Association for Computational Linguistics. URL `https://aclanthology.org/D13-1160`.

Ben Bogin, Jonathan Berant, and Matt Gardner. Representing schema structure with graph neural networks for text-to-SQL parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4560–4565, Florence, Italy, July 2019a. Association for Computational Linguistics. doi: 10.18653/v1/P19-1448. URL `https://www.aclweb.org/anthology/P19-1448`.

Ben Bogin, Matt Gardner, and Jonathan Berant. Global Reasoning over Database Structures for Text-to-SQL Parsing. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3659–3664, Hong Kong, China, nov 2019b. Association for Computational Linguistics. doi: 10.18653/v1/D19-1378. URL `https://www.aclweb.org/anthology/D19-1378`.

Paweł Budzianowski, Tsung-Hsien Wen, Bo-Hsiang Tseng, Iñigo Casanueva, Stefan Ultes, Osman Ramadan, and Milica Gašić. MultiWOZ - a large-scale multi-domain Wizard-of-Oz dataset for task-oriented dialogue modelling. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 5016–5026, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1547. URL `https://aclanthology.org/D18-1547`.

Ruichu Cai, Jinjie Yuan, Boyan Xu, and Zhifeng Hao. Sadga: Structure-aware dual graph aggregation network for text-to-sql, 2021.

Marco Antonio Calijorne Soares and Fernando Silva Parreiras. A literature review on question answering techniques, paradigms and systems. *Journal of King Saud University - Computer and Information Sciences*, 32(6): 635–646, 2020. ISSN 1319-1578. doi: https://doi.org/10.1016/j.jksuci.2018.08.005. URL `https://www.sciencedirect.com/science/article/pii/S131915781830082X`.

Ruisheng Cao, Lu Chen, Zhi Chen, Yanbin Zhao, Su Zhu, and Kai Yu. LGESQL: Line graph enhanced text-to-SQL model with mixed local and non-local relations. 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 2541–2555, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.198. URL `https://aclanthology.org/2021.acl-long.198`.

Sanxing Chen, Aidan San, Xiaodong Liu, and Yangfeng Ji. A tale of two linkings: Dynamically gating between schema linking and structural linking for text-to-SQL parsing. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 2900–2912, Barcelona, Spain (Online), December 2020a. International Committee on Computational Linguistics. doi: 10.18653/v1/2020.coling-main.260. URL `https://aclanthology.org/2020.coling-main.260`.

Xinyun Chen, Chen Liang, Adams Wei Yu, Dawn Song, and Denny Zhou. Compositional generalization via neural-symbolic stack machines. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1690–1701. Curran Associates, Inc., 2020b. URL `https://proceedings.neurips.cc/paper/2020/file/12b1e42dc0746f22cf361267de07073f-Paper.pdf`.

Jianpeng Cheng, Siva Reddy, Vijay Saraswat, and Mirella Lapata. Learning structured natural language representations for semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 44–55, Vancouver, Canada, July 2017a. Association for Computational Linguistics. doi: 10.18653/v1/P17-1005. URL `https://aclanthology.org/P17-1005`.

Jianpeng Cheng, Siva Reddy, Vijay Saraswat, and Mirella Lapata. Learning structured natural language representations for semantic parsing. In *55th Annual Meeting of the Association for Computational Linguistics, ACL 2017*, pages 44–55. Association for Computational Linguistics (ACL), 2017b.

Minseok Cho, Reinald Kim Amplayo, Seung-won Hwang, and Jonghyuck Park. Adversarial tableqa: Attention supervision for question answering on tables. In Jun Zhu and Ichiro Takeuchi, editors, *Proceedings of The 10th Asian Conference on Machine Learning*, volume 95 of *Proceedings of Machine Learning Research*, pages 391–406. PMLR, 14–16 Nov 2018. URL `https://proceedings.mlr.press/v95/cho18a.html`.

Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. Electra: Pre-training text encoders as discriminators rather than generators, 2020. URL `https://arxiv.org/abs/2003.10555`.

Henry Conklin, Bailin Wang, Kenny Smith, and Ivan Titov. Meta-learning to compositionally generalize. 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 3322–3335, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.258. URL `https://aclanthology.org/2021.acl-long.258`.

Marie-Catherine de Marnee and Christopher D. Manning. Stanford typed dependencies manual. 2016. URL `https://aclanthology.org/2020.coling-main.34`.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding.

In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL `https://www.aclweb.org/anthology/N19-1423`.

Li Dong and Mirella Lapata. Language to logical form with neural attention. In *54th Annual Meeting of the Association for Computational Linguistics*, pages 33–43. Association for Computational Linguistics (ACL), 2016.

Li Dong and Mirella Lapata. Coarse-to-Fine Decoding for Neural Semantic Parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 731–742, Stroudsburg, PA, USA, 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1068. URL `http://aclweb.org/anthology/P18-1068`.

Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. HotFlip: White-box adversarial examples for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 31–36, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-2006. URL `https://www.aclweb.org/anthology/P18-2006`.

Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. Improving text-to-SQL evaluation methodology. pages 351–360, July 2018. doi: 10.18653/v1/P18-1033. URL `https://aclanthology.org/P18-1033`.

Daniel Furrer, Marc van Zee, Nathan Scales, and Nathanael Schärli. Compositional generalization in semantic parsing: Pre-training vs. specialized architectures. *CoRR*, abs/2007.08970, 2020. URL `https://arxiv.org/abs/2007.08970`.

Yujian Gan, Matthew Purver, and John R. Woodward. A review of cross-domain text-to-SQL models. In *Proceedings of the 1st Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 10th International Joint Conference on Natural Language Processing: Student Research Workshop*, pages 108–115, Suzhou, China, December 2020. Association for Computational Linguistics. URL `https://aclanthology.org/2020.aacl-srw.16`.

Yujian Gan, Xinyun Chen, Qiuping Huang, Matthew Purver, John R. Woodward, Jinxia Xie, and Pengsheng Huang. Towards robustness of text-to-SQL models against synonym substitution. 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 2505–2515, Online, August 2021a. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.195. URL `https://aclanthology.org/2021.acl-long.195`.

Yujian Gan, Xinyun Chen, and Matthew Purver. Exploring underexplored limitations of cross-domain text-to-sql generalization. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2021b.

Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R. Woodward, John Drake, and Qiaofu Zhang. Natural sql: Making sql easier to infer from natural language specifications, 2021c.

Yujian Gan, Xinyun Chen, Qiuping Huang, and Matthew Purver. Measuring and improving compositional generalization in text-to-sql via component alignment, 2022. URL `https://arxiv.org/abs/2205.02054`.

Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson Liu, Matthew Peters, Michael Schmitz, and Luke Zettlemoyer. AllenNLP: A Deep Semantic Natural Language Processing Platform. mar 2018. URL `http://arxiv.org/abs/1803.07640`.

F.A. Gers. Learning to forget: continual prediction with LSTM. In *9th International Conference on Artificial Neural Networks: ICANN '99*, volume 1999, pages 850–855. IEE, 1999. ISBN 0 85296 721 7. doi: 10.1049/cp:19991218. URL `https://digital-library.theiet.org/content/conferences/10.1049/cp{_}19991218`.

Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1263–1272. PMLR, 06–11 Aug 2017. URL `https://proceedings.mlr.press/v70/gilmer17a.html`.

Alessandra Giordani and Alessandro Moschitti. Automatic Generation and Reranking of SQL-derived Answers to NL Questions. In *Proceedings of the Second International Conference on Trustworthy Eternal Systems via Evolving Software, Data and Knowledge*, pages 59–76, 2012. URL `https://doi.org/10.1007/978-3-642-45260-4{_}5`.

Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4524–4535, Florence, Italy, jul 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1444. URL `https://www.aclweb.org/anthology/P19-1444`.

Pengcheng He, Yi Mao, Kaushik Chakrabarti, and Weizhu Chen. X-SQL: REINFORCE CONTEXT INTO SCHEMA REPRESENTATION. *https://www.microsoft.com/en-us/research/uploads/prod/2019/03/X_SQL-5c7db555d760f.pdf*, 2019.

Gary G Hendrix, Earl D Sacerdoti, Daniel Sagalowicz, and Jonathan Slocum. Developing a natural language interface to complex data. *ACM Transactions on Database Systems (TODS)*, 3(2):105–147, 1978.

Jonathan Herzig and Jonathan Berant. Span-based semantic parsing for compositional generalization. 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 908–921, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.74. URL `https://aclanthology.org/2021.acl-long.74`.

Jonathan Herzig, Peter Shaw, Ming-Wei Chang, Kelvin Guu, Panupong Pasupat, and Yuan Zhang. Unlocking compositional generalization in pre-trained models using intermediate representations. *CoRR*, abs/2104.07478, 2021a. URL `https://arxiv.org/abs/2104.07478`.

Jonathan Herzig, Peter Shaw, Ming-Wei Chang, Kelvin Guu, Panupong Pasupat, and Yuan Zhang. Unlocking compositional generalization in pre-trained models using intermediate representations. *arXiv preprint arXiv:2104.07478*, 2021b.

Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, nov 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL `http://www.mitpressjournals.org/doi/10.1162/neco.1997.9.8.1735`.

J J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America*, 79(8):2554–8, apr 1982. ISSN 0027-8424. doi: 10.1073/pnas.79.8.2554.

URL http://www.ncbi.nlm.nih.gov/pubmed/6953413http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC346238.

Binyuan Hui, Ruiying Geng, Lihan Wang, Bowen Qin, Bowen Li, Jian Sun, and Yongbin Li. S^{2}sql: Injecting syntax to question-schema interaction graph encoder for text-to-sql parsers, 2022. URL https://arxiv.org/abs/2203.06958.

Radu Cristian Alexandru Iacob, Florin Brad, Elena-Simona Apostol, Ciprian-Octavian Truică, Ionel Alexandru Hosu, and Traian Rebedea. Neural approaches for natural language interfaces to databases: A survey. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 381–395, Barcelona, Spain (Online), December 2020. International Committee on Computational Linguistics. doi: 10.18653/v1/2020.coling-main.34. URL https://aclanthology.org/2020.coling-main.34.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. Learning a Neural Semantic Parser from User Feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 963–973, 2017. ISBN 9781945626753. doi: 10.18653/v1/P17-1089. URL http://arxiv.org/abs/1704.08760.

Sujay Kumar Jauhar, Peter Turney, and Eduard Hovy. Tables as semi-structured knowledge for question answering. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 474–483, 2016.

Robin Jia and Percy Liang. Data recombination for neural semantic parsing, 2016. URL https://arxiv.org/abs/1606.03622.

Nengzheng Jin, Joanna Siebert, Dongfang Li, and Qingcai Chen. A survey on table question answering: Recent advances. *arXiv preprint arXiv:2207.05270*, 2022.

Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the Limits of Language Modeling. feb 2016. URL http://arxiv.org/abs/1602.02410.

Aishwarya Kamath and Rajarshi Das. A survey on semantic parsing, 2018. URL https://arxiv.org/abs/1812.00978.

Divyansh Kaushik, Eduard Hovy, and Zachary Lipton. Learning the difference that makes a difference with counterfactually-augmented data. In *In-*

ternational Conference on Learning Representations, 2020. URL `https://openreview.net/forum?id=SklgsONFvr`.

J-D Kim, Tomoko Ohta, Yuka Tateisi, and Jun'ichi Tsujii. Genia corpus—a semantically annotated corpus for bio-textmining. *Bioinformatics*, 19(suppl_1): i180–i182, 2003.

Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. Neural Semantic Parsing with Type Constraints for Semi-Structured Tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1516–1526, Stroudsburg, PA, USA, 2017. Association for Computational Linguistics. doi: 10.18653/v1/D17-1160. URL `http://aclweb.org/anthology/D17-1160`.

Tom Kwiatkowksi, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. Inducing probabilistic ccg grammars from logical form with higher-order unification. In *Proceedings of the 2010 conference on empirical methods in natural language processing*, pages 1223–1233, 2010.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Matthew Kelcey, Jacob Devlin, Kenton Lee, Kristina N. Toutanova, Llion Jones, Ming-Wei Chang, Andrew Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: a benchmark for question answering research. *Transactions of the Association of Computational Linguistics*, 2019.

Brenden Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2873–2882. PMLR, 10–15 Jul 2018. URL `https://proceedings.mlr.press/v80/lake18a.html`.

Dongjun Lee. Clause-Wise and Recursive Decoding for Complex and Cross-Domain Text-to-SQL Generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6045–6051, Hong Kong, China, nov 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1624. URL `https://www.aclweb.org/anthology/D19-1624`.

Wenqiang Lei, Weixin Wang, Zhixin Ma, Tian Gan, Wei Lu, Min-Yen Kan, and Tat-Seng Chua. Re-examining the Role of Schema Linking in Text-to-SQL. In *Proceedings of the 2020 Conference on Empirical Methods in*

Natural Language Processing (EMNLP), pages 6943–6954, Stroudsburg, PA, USA, nov 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.564. URL `https://www.aclweb.org/anthology/2020.emnlp-main.564`.

Fei Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84, sep 2014. ISSN 21508097. doi: 10.14778/2735461.2735468. URL `http://dl.acm.org/citation.cfm?doid=2735461.2735468`.

Fei Li, Tianyin Pan, and Hosagrahar V. Jagadish. Schema-free SQL. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD '14*, pages 1051–1062, New York, New York, USA, 2014. ACM Press. ISBN 9781450323765. doi: 10.1145/2588555.2588571. URL `http://dl.acm.org/citation.cfm?doid=2588555.2588571`.

Jingjing Li, Wenlu Wang, Wei Shinn Ku, Yingtao Tian, and Haixun Wang. SpatialNLI: A spatial domain natural language interface to databases using spatial comprehension. In *GIS: Proceedings of the ACM International Symposium on Advances in Geographic Information Systems*, pages 339–348, New York, NY, USA, nov 2019. Association for Computing Machinery. ISBN 9781450369091. doi: 10.1145/3347146.3359069. URL `https://dl.acm.org/doi/10.1145/3347146.3359069`.

Linyang Li, Ruotian Ma, Qipeng Guo, Xiangyang Xue, and Xipeng Qiu. BERT-ATTACK: Adversarial Attack Against BERT Using BERT. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6193–6202, Stroudsburg, PA, USA, apr 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.500. URL `https://www.aclweb.org/anthology/2020.emnlp-main.500`.

Yunyao Li, Huahai Yang, and H V Jagadish. Constructing a Generic Natural Language Interface for an XML Database. In Yannis Ioannidis, Marc H Scholl, Joachim W Schmidt, Florian Matthes, Mike Hatzopoulos, Klemens Boehm, Alfons Kemper, Torsten Grust, and Christian Boehm, editors, *Advances in Database Technology - EDBT 2006*, pages 737–754, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-32961-9.

Xi Victoria Lin, Richard Socher, and Caiming Xiong. Bridging Textual and Tabular Data for Cross-Domain Text-to-SQL Semantic Parsing. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4870–4888, Online, nov 2020. Association for Computational Linguistics. URL `https://www.aclweb.org/anthology/2020.findings-emnlp.438`.

Qian Liu, Shengnan An, Jian-Guang Lou, Bei Chen, Zeqi Lin, Yan Gao, Bin Zhou, Nanning Zheng, and Dongmei Zhang. Compositional generalization by learning analytical expressions. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 11416–11427. Curran Associates, Inc., 2020. URL `https://proceedings.neurips.cc/paper/2020/file/83adc9225e4deb67d7ce42d58fe5157c-Paper.pdf`.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective Approaches to Attention-based Neural Machine Translation. aug 2015. URL `http://arxiv.org/abs/1508.04025`.

Qin Lyu, Kaushik Chakrabarti, Shobhit Hathi, Souvik Kundu, Jianwen Zhang, and Zheng Chen. Hybrid Ranking Network for Text-to-SQL. *https://www.microsoft.com/en-us/research/uploads/prod/2020/03/HydraNet_ 20200311-5e69612887fcb.pdf*, mar 2020.

Jianqiang Ma, Zeyu Yan, Shuai Pang, Yang Zhang, and Jianping Shen. Mention extraction and linking for SQL query generation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6936–6942, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.563. URL `https://aclanthology.org/2020.emnlp-main.563`.

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018.

Qingkai Min, Yuefeng Shi, and Yue Zhang. A pilot study for Chinese SQL semantic parsing. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3652–3658, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1377. URL `https://aclanthology.org/D19-1377`.

John Morris, Eli Lifland, Jin Yong Yoo, Jake Grigsby, Di Jin, and Yanjun Qi. TextAttack: A Framework for Adversarial Attacks, Data Augmentation, and Adversarial Training in NLP. In *Proceedings of the 2020 Confer-*

ence on Empirical Methods in Natural Language Processing: System Demonstrations, pages 119–126, Stroudsburg, PA, USA, nov 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-demos.16. URL `https://www.aclweb.org/anthology/2020.emnlp-demos.16`.

Nikola Mrkšić, Diarmuid Ó Séaghdha, Blaise Thomson, Milica Gašić, Lina M. Rojas-Barahona, Pei-Hao Su, David Vandyke, Tsung-Hsien Wen, and Steve Young. Counter-fitting word vectors to linguistic constraints. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 142–148, San Diego, California, June 2016. Association for Computational Linguistics. doi: 10.18653/v1/N16-1018. URL `https://www.aclweb.org/anthology/N16-1018`.

Inbar Oren, Jonathan Herzig, Nitish Gupta, Matt Gardner, and Jonathan Berant. Improving compositional generalization in semantic parsing. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 2482–2495, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.225. URL `https://www.aclweb.org/anthology/2020.findings-emnlp.225`.

Panupong Pasupat and Percy Liang. Compositional Semantic Parsing on Semi-Structured Tables. 2015. ISSN 1572-9575. doi: 10.3115/v1/P15-1142. URL `http://arxiv.org/abs/1508.00305`.

Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1162. URL `https://www.aclweb.org/anthology/D14-1162`.

Hoifung Poon and Pedro Domingos. Unsupervised semantic parsing. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 1–10, Singapore, August 2009. Association for Computational Linguistics. URL `https://aclanthology.org/D09-1001`.

Hoifung Poon and Pedro Domingos. Unsupervised ontology induction from text. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 296–305, Uppsala, Sweden, July 2010. Association for Computational Linguistics. URL `https://aclanthology.org/P10-1031`.

Ana-Maria Popescu, Alex Armanasu, Oren Etzioni, David Ko, and Alexander Yates. Modern natural language interfaces to databases: composing statistical parsing with semantic tractability. In *Proceedings of the 20th international conference on Computational Linguistics - COLING '04*, pages 141–es, Morristown, NJ, USA, 2004. Association for Computational Linguistics. doi: 10.3115/1220355.1220376. URL `http://portal.acm.org/citation.cfm?doid=1220355.1220376`.

Bowen Qin, Binyuan Hui, Lihan Wang, Min Yang, Jinyang Li, Binhua Li, Ruiying Geng, Rongyu Cao, Jian Sun, Luo Si, Fei Huang, and Yongbin Li. A survey on text-to-sql parsing: Concepts, methods, and future directions, 2022. URL `https://arxiv.org/abs/2208.13629`.

Minghui Qiu, Feng-Lin Li, Siyu Wang, Xing Gao, Yan Chen, Weipeng Zhao, Haiqing Chen, Jun Huang, and Wei Chu. AliMe Chat: A Sequence to Sequence and Rerank based Chatbot Engine. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 498–503, Stroudsburg, PA, USA, 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-2079. URL `http://aclweb.org/anthology/P17-2079`.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing, 2017. URL `https://arxiv.org/abs/1704.07535`.

Karthik Radhakrishnan, Arvind Srikantan, and Xi Victoria Lin. ColloQL: Robust Cross-Domain Text-to-SQL Over Search Queries, oct 2020. URL `https://arxiv.org/abs/2010.09927`.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2019. URL `https://arxiv.org/abs/1910.10683`.

Ohad Rubin and Jonathan Berant. SmBoP: Semi-autoregressive bottom-up semantic parsing. pages 311–324, June 2021. doi: 10.18653/v1/2021.naacl-main.29. URL `https://aclanthology.org/2021.naacl-main.29`.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, oct 1986. ISSN 0028-0836. doi: 10.1038/323533a0. URL `http://www.nature.com/articles/323533a0`.

Shiori Sagawa, Pang Wei Koh, Tatsunori B. Hashimoto, and Percy Liang. Distributionally robust neural networks for group shifts: On the importance of regularization for worst-case generalization, 2020.

H Sak, A Senior, F Beaufays Fifteenth annual conference of The, and Undefined 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. *isca-speech.org*, 2014. URL `https://www.isca-speech.org/archive/interspeech{_}2014/i14{_}0338.html`.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009. doi: 10.1109/TNN.2008.2005605.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. Picard: Parsing incrementally for constrained auto-regressive decoding from language models, 2021.

M. Schuster and K.K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997. ISSN 1053587X. doi: 10.1109/78.650093. URL `http://ieeexplore.ieee.org/document/650093/`.

R. S. Scowen. Extended BNF — A generic base standard. In *Proceedings 1993 Software Engineering Standards Symposium*, pages 25–34, 1993. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.124.9111`.

Peter Shaw, Ming-Wei Chang, Panupong Pasupat, and Kristina Toutanova. Compositional generalization and natural language variation: Can a semantic parsing approach handle both? 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 922–938, Online, August 2021a. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.75. URL `https://aclanthology.org/2021.acl-long.75`.

Peter Shaw, Ming-Wei Chang, Panupong Pasupat, and Kristina Toutanova. Compositional generalization and natural language variation: Can a semantic parsing approach handle both? 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 922–938, Online, August 2021b. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.75. URL `https://aclanthology.org/2021.acl-long.75`.

Peng Shi, Patrick Ng, Zhiguo Wang, Henghui Zhu, Alexander Hanbo Li, Jun Wang, Cicero Nogueira dos Santos, and Bing Xiang. Learning contextual representations for semantic parsing with generation-augmented pre-training.

Proceedings of the AAAI Conference on Artificial Intelligence, 35(15):13806–13814, May 2021. URL `https://ojs.aaai.org/index.php/AAAI/article/view/17627`.

Ram Shankar Siva Kumar, Magnus Nyström, John Lambert, Andrew Marshall, Mario Goertzel, Andi Comissoneru, Matt Swann, and Sharon Xia. Adversarial machine learning-industry perspectives. In *2020 IEEE Security and Privacy Workshops (SPW)*, pages 69–75, 2020. doi: 10.1109/SPW50608.2020.00028.

Robyn Speer and Catherine Havasi. Representing General Relational Knowledge in ConceptNet 5. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)*, pages 3679–3686, Istanbul, Turkey, may 2012a. European Language Resources Association (ELRA). URL `http://www.lrec-conf.org/proceedings/lrec2012/pdf/1072_Paper.pdf`.

Robyn Speer and Catherine Havasi. Representing general relational knowledge in conceptnet 5. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC-2012)*, pages 3679–3686, 2012b.

Alane Suhr, Ming-Wei Chang, Peter Shaw, and Kenton Lee. Exploring unexplored generalization challenges for cross-database semantic parsing. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8372–8388, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.742. URL `https://www.aclweb.org/anthology/2020.acl-main.742`.

Ningyuan Sun, Xuefeng Yang, and Yunfeng Liu. Tableqa: a large-scale chinese text-to-sql dataset for table-aware sql generation. *arXiv preprint arXiv:2006.06434*, 2020.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3104–3112. Curran Associates, Inc., 2014. URL `http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf`.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks, 2013. URL `https://arxiv.org/abs/1312.6199`.

Alon Talmor, Ori Yoran, Amnon Catav, Dan Lahav, Yizhong Wang, Akari Asai, Gabriel Ilharco, Hannaneh Hajishirzi, and Jonathan Berant. Multimodal{qa}:

complex question answering over text, tables and images. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=ee6W5UgQLa`.

Sinan Tan, Mengmeng Ge, Di Guo, Huaping Liu, and Fuchun Sun. Knowledge-based embodied question answering, 2021.

Lappoon R Tang and Raymond J Mooney. Automated Construction of Database Interfaces: Intergrating Statistical and Relational Learning for Semantic Parsing. In *2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, pages 133–141, 2000. URL `http://www.aclweb.org/anthology/W00-1317`.

Yasufumi Taniguchi, Hiroki Nakayama, Kubo Takahiro, and Jun Suzuki. An investigation between schema linking and text-to-sql performance, 2021.

Marjorie Templeton and John F Burger. Problems in natural-language interface to dbms with examples from eufid. In *First Conference on Applied Natural Language Processing*, pages 3–16, 1983.

Cynthia Thompson. Acquiring word-meaning mappings for natural language interfaces. *Journal of Artificial Intelligence Research*, 18:1–44, 2003.

Ashish Vaswani. Attention Is All You Need. (Nips), 2017.

Jesse Vig. A multiscale visualization of attention in the transformer model. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 37–42, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-3007. URL `https://www.aclweb.org/anthology/P19-3007`.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7567–7578, Online, jul 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.677. URL `https://www.aclweb.org/anthology/2020.acl-main.677`.

Bailin Wang, Mirella Lapata, and Ivan Titov. Structured reordering for modeling latent alignments in sequence transduction. In *Thirty-Fifth Conference on Neural Information Processing Systems*, 2021. URL `https://openreview.net/forum?id=X2Cxixkcpx`.

Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov, and Rishabh Singh. Robust Text-to-SQL Generation with Execution-Guided Decoding. 2018. URL `http://arxiv.org/abs/1807.03100`.

Daniel C Wang, Andrew W Appel, Jeffrey L Korn, and Christopher S Serra. The zephyr abstract syntax description language. In *DSL*, volume 97, pages 17–17, 1997.

Zhen Wang. Modern question answering datasets and benchmarks: A survey, 2022. URL `https://arxiv.org/abs/2206.15030`.

David H D Warren and Fernando C N Pereira. An Efficient Easily Adaptable System for Interpreting Natural Language Queries. *Comput. Linguist.*, 8(3-4):110–122, jul 1982. ISSN 0891-2017. URL `http://dl.acm.org/citation.cfm?id=972942.972944`.

Jason Wei and Kai Zou. EDA: Easy data augmentation techniques for boosting performance on text classification tasks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6382–6388, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1670. URL `https://www.aclweb.org/anthology/D19-1670`.

Wikipedia-contributors. Machine learning — {Wikipedia}{,} The Free Encyclopedia, 2019. URL `https://en.wikipedia.org/w/index.php?title=Machine{_}learning{&}oldid=896127124`.

Wikipedia contributors. Graph neural network — Wikipedia, the free encyclopedia, 2022a. URL `https://en.wikipedia.org/w/index.php?title=Graph_neural_network&oldid=1074555095`. [Online; accessed 12-March-2022].

Wikipedia contributors. Intermediate representation — Wikipedia, the free encyclopedia, 2022b. URL `https://en.wikipedia.org/w/index.php?title=Intermediate_representation&oldid=1088248902`. [Online; accessed 26-October-2022].

Wikipedia contributors. Sql — Wikipedia, the free encyclopedia, 2022c. URL `https://en.wikipedia.org/w/index.php?title=SQL&oldid=1116513499`. [Online; accessed 18-October-2022].

Wikipedia contributors. Adversarial machine learning — Wikipedia, the free encyclopedia, 2022d. URL `https://en.wikipedia.org/w/index.php?title=`

Adversarial_machine_learning&oldid=1116514435. [Online; accessed 18-October-2022].

Yuk Wah Wong and Raymond Mooney. Learning for semantic parsing with statistical machine translation. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 439–446, 2006.

Minjoon Seo Wonseok Hwang, Jinyeung Yim, Seunghyun Park. A Comprehensive Exploration on WikiSQL with Table-Aware Word Contextualization, 2019. URL https://arxiv.org/abs/1902.01069.

W.A. Woods. Semantics and Quantification in Natural Language Question Answering. *Advances in Computers*, 17:1–87, jan 1978. ISSN 0065-2458. doi: 10.1016/S0065-2458(08)60390-3. URL https://www.sciencedirect.com/science/article/pii/S0065245808603903.

William A Woods. Progress in natural language understanding: an application to lunar geology. In *Proceedings of the June 4-8, 1973, national computer conference and exposition*, pages 441–450, 1973.

Chunyang Xiao, Marc Dymetman, and Claire Gardent. Sequence-based Structured Prediction for Semantic Parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1341–1350, Stroudsburg, PA, USA, 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1127. URL http://aclweb.org/anthology/P16-1127.

Tianbao Xie, Chen Henry Wu, Peng Shi, Ruiqi Zhong, Torsten Scholak, Michihiro Yasunaga, Chien-Sheng Wu, Ming Zhong, Pengcheng Yin, Sida I. Wang, Victor Zhong, Bailin Wang, Chengzu Li, Connor Boyle, Ansong Ni, Ziyu Yao, Dragomir Radev, Caiming Xiong, Lingpeng Kong, Rui Zhang, Noah A. Smith, Luke Zettlemoyer, and Tao Yu. Unifiedskg: Unifying and multi-tasking structured knowledge grounding with text-to-text language models, 2022. URL https://arxiv.org/abs/2201.05966.

Hongvu Xiong and Ruixiao Sun. Transferable Natural Language Interface to Structured Queries Aided by Adversarial Generation. In *2019 IEEE 13th International Conference on Semantic Computing (ICSC)*, pages 255–262. IEEE, jan 2019. ISBN 978-1-5386-6783-5. doi: 10.1109/ICOSC.2019.8665499. URL https://arxiv.org/abs/1812.01245https://ieeexplore.ieee.org/document/8665499/.

Peng Xu, Dhruv Kumar, Wei Yang, Wenjie Zi, Keyi Tang, Chenyang Huang, Jackie Chi Kit Cheung, Simon J.D. Prince, and Yanshuai Cao. Optimizing deeper transformers on small datasets. 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 2089–2102, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.163. URL `https://aclanthology.org/2021.acl-long.163`.

Xiaojun Xu, Chang Liu, and Dawn Song. SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning. *Mathematics of Computation*, 22(103):651, nov 2017. ISSN 00255718. doi: 10.2307/2004542. URL `http://arxiv.org/abs/1711.04436`.

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. SQLizer: Query Synthesis from Natural Language. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM*, pages 63:1—-63:26, oct 2017. URL `http://doi.org/10.1145/3133887`.

Pengcheng Yin and Graham Neubig. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Stroudsburg, PA, USA, 2017a. Association for Computational Linguistics. doi: 10.18653/v1/P17-1041. URL `http://aclweb.org/anthology/P17-1041`.

Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation, 2017b. URL `https://arxiv.org/abs/1704.01696`.

Pengcheng Yin, Hao Fang, Graham Neubig, Adam Pauls, Emmanouil Antonios Platanios, Yu Su, Sam Thomson, and Jacob Andreas. Compositional generalization for neural semantic parsing via span-level supervised attention. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2810–2823, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.225. URL `https://aclanthology.org/2021.naacl-main.225`.

Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev. TypeSQL: Knowledge-based type-aware neural text-to-SQL generation. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 588–594, New Orleans, Louisiana, jun . Association for Computational Linguistics. doi: 10.18653/v1/N18-2093. URL `https://www.aclweb.org/anthology/N18-2093`.

Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. SyntaxSQLNet: Syntax tree networks for complex and cross-domain text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1653–1663, Brussels, Belgium, October-November 2018a. Association for Computational Linguistics. doi: 10.18653/v1/D18-1193. URL `https://www.aclweb.org/anthology/D18-1193`.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium, October-November 2018b. Association for Computational Linguistics. doi: 10.18653/v1/D18-1425. URL `https://www.aclweb.org/anthology/D18-1425`.

Tao Yu, Rui Zhang, Heyang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, et al. Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1962–1979, 2019a.

Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, Suyi Li, Heyang Er, Irene Li, Bo Pang, Tao Chen, Emily Ji, Shreya Dixit, David Proctor, Sungrok Shim, Jonathan Kraft, Vincent Zhang, Caiming Xiong, Richard Socher, and Dragomir Radev. SParC: Cross-domain semantic parsing in context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4511–4523, Florence, Italy, July 2019b. Association for Computational Linguistics. doi: 10.18653/v1/P19-1443. URL `https://aclanthology.org/P19-1443`.

Tao Yu, Chien-Sheng Wu, Xi Victoria Lin, Bailin Wang, Yi Chern Tan, Xinyi Yang, Dragomir Radev, Richard Socher, and Caiming Xiong. Grappa: Grammar-augmented pre-training for table semantic parsing, 2021.

John M Zelle and Raymond J Mooney. Learning to Parse Database Queries Using Inductive Logic Programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2*, pages 1050–1055, 1996a. URL `http://dl.acm.org/citation.cfm?id=1864519.1864543`.

John M Zelle and Raymond J Mooney. Learning to parse database queries using inductive logic programming. In *Proceedings of the national conference on artificial intelligence*, pages 1050–1055, 1996b.

Jichuan Zeng, Xi Victoria Lin, Steven C.H. Hoi, Richard Socher, Caiming Xiong, Michael Lyu, and Irwin King. Photon: A Robust Cross-Domain Text-to-SQL System. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 204–214, Stroudsburg, PA, USA, jul 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-demos.24. URL `https://www.aclweb.org/anthology/2020.acl-demos.24`.

Luke Zettlemoyer and Michael Collins. Online learning of relaxed ccg grammars for parsing to logical form. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 678–687, 2007.

Luke S Zettlemoyer and Michael Collins. Learning to map sentences to logical form: structured classification with probabilistic categorial grammars. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*, pages 658–666, 2005.

Rui Zhang, Tao Yu, Heyang Er, Sungrok Shim, Eric Xue, Xi Victoria Lin, Tianze Shi, Caiming Xiong, Richard Socher, and Dragomir Radev. Editing-based SQL query generation for cross-domain context-dependent questions. pages 5338–5349, November 2019. doi: 10.18653/v1/D19-1537. URL `https://aclanthology.org/D19-1537`.

Ruiqi Zhong, Tao Yu, and Dan Klein. Semantic evaluation for text-to-sql with distilled test suite. In *The 2020 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2020a.

Victor Zhong, Caiming Xiong, and Richard Socher. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *CoRR*, abs/1709.0, 2017.

Victor Zhong, Mike Lewis, Sida I Wang, and Luke Zettlemoyer. Grounded adaptation for zero-shot executable semantic parsing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6869–6882, 2020b.

Yi Zhu, Yiwei Zhou, and Menglin Xia. Generating Semantically Valid Adversarial Questions for TableQA, may 2020. URL `http://arxiv.org/abs/2005.12696`.

Appendix A

Domain Konwledge

A domain means a certain type of application scenarios; for example, the Spider benchmark includes various distinct domains such as geography and university. Cross-domain text-to-SQL research aims to build a text-to-SQL model that can generate correct SQL queries and generalize to different domains. Therefore, one main challenge of cross-domain text-to-SQL generalization is to understand different knowledge required by different domains. For example, the university domain usually needs the knowledge of different job titles and genders, while the geography domain emphasizes more on the knowledge of places instead of people.

Different SQL databases could require very different domain knowledge. As shown in [Suhr et al., 2020], the state-of-the-art models on Spider achieve much worse performance on earlier SQL benchmarks such as ATIS and GeoQuery [Iyer et al., 2017, Zelle and Mooney, 1996a]. However, we argue that the failure of generalization is expected to some extent, because without seeing in-domain examples, some domain knowledge required by these datasets is even hard to infer for experienced programmers. For example, we asked five computer science graduate students to write the SQL query for the question `'how many major cities are there?'` in GeoQuery, but none of them gave the correct answer. This question requires the domain knowledge that `major` means `'population > 150000'`, which is hard to infer without looking at GeoQuery training set. Therefore, while acquiring general-purpose domain knowledge is also important, we believe that the failure of generalization to questions requiring similar domain knowledge to the training set could be more problematic, which motivates our design of Spider-DK benchmark.