Data Augmentation with Hierarchical SQL-to-Question Generation for Cross-domain Text-to-SQL Parsing

Kun Wu[1]*, **Lijie Wang**[2], **Zhenghua Li**[1], **Ao Zhang**[2],
Xinyan Xiao[2], **Hua Wu**[2], **Min Zhang**[1], **Haifeng Wang**[2]
1. Institute of Artificial Intelligence, School of Computer Science and Technology,
Soochow University, Suzhou, China
2. Baidu Inc, Beijing, China
kwu@stu.suda.edu.cn, {zhli13,minzhang}@suda.edu.cn
{wanglijie,zhangao,xiaoxinyan,wu_hua,wanghaifeng}@baidu.com

Abstract

Data augmentation has attracted a lot of research attention in the deep learning era for its ability in alleviating data sparseness. The lack of labeled data for unseen evaluation databases is exactly the major challenge for cross-domain text-to-SQL parsing. Previous works either require human intervention to guarantee the quality of generated data, or fail to handle complex SQL queries. This paper presents a simple yet effective data augmentation framework. First, given a database, we automatically produce a large number of SQL queries based on an abstract syntax tree grammar. For better distribution matching, we require that at least 80% of SQL patterns in the training data are covered by generated queries. Second, we propose a hierarchical SQL-to-question generation model to obtain high-quality natural language questions, which is the major contribution of this work. Finally, we design a simple sampling strategy that can greatly improve training efficiency given large amounts of generated data. Experiments on three cross-domain datasets, i.e., WikiSQL and Spider in English, and DuSQL in Chinese, show that our proposed data augmentation framework can consistently improve performance over strong baselines, and the hierarchical generation component is the key for the improvement.

1 Introduction

Given a natural language (NL) question and a relational database (DB), the text-to-SQL parsing task aims to produce a legal and executable SQL query to get the correct answer (Date and Darwen, 1997), as depicted in Figure 1. A DB usually consists of multiple tables interconnected via foreign keys.

Early research on text-to-SQL parsing mainly focuses on the in-domain setting (Li and Jagadish, 2014; Iyer et al., 2017; Yaghmazadeh et al., 2017), where all question/SQL pairs of train/dev/test sets are generated against the same DB. In order to deal with the more realistic setting where DBs in the evaluation phase are unseen in the training data, researchers propose several cross-domain datasets, such as WikiSQL (Zhong et al., 2017) and Spider (Yu et al., 2018b) in English, and DuSQL (Wang et al., 2020b) in Chinese. All three datasets adopt the DB-level data splitting, meaning that a DB and all its corresponding question/SQL pairs can appear in only one of the train/dev/test sets.

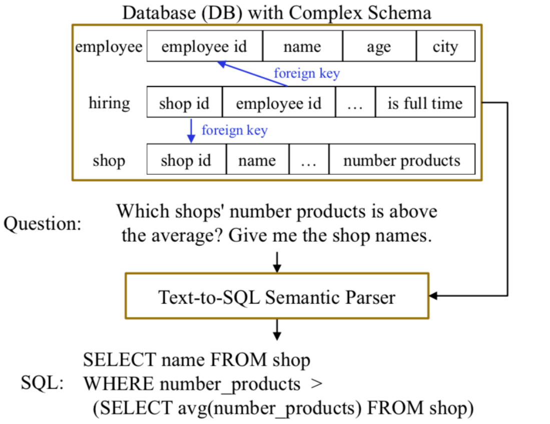


Figure 1: An example of the text-to-SQL parsing task.

Cross-domain text-to-SQL parsing has two major challenges. First, unseen DBs usually introduce new schemas, such as new table/column names and unknown semantics of inter-table relationships. Therefore, it is crucial for a parsing model to have strong generalization ability. The second challenge is that the scale of labeled data is quite small for such a complex task, since it is extremely difficult to construct DBs and manually annotate corresponding question/SQL pairs. For example, the Spider dataset has only 200 DBs and 10K question/SQL pairs in total.

To deal with the first challenge, many previous works focus on how to better encode the matching

[*] Work done during an internship at Baidu Inc.

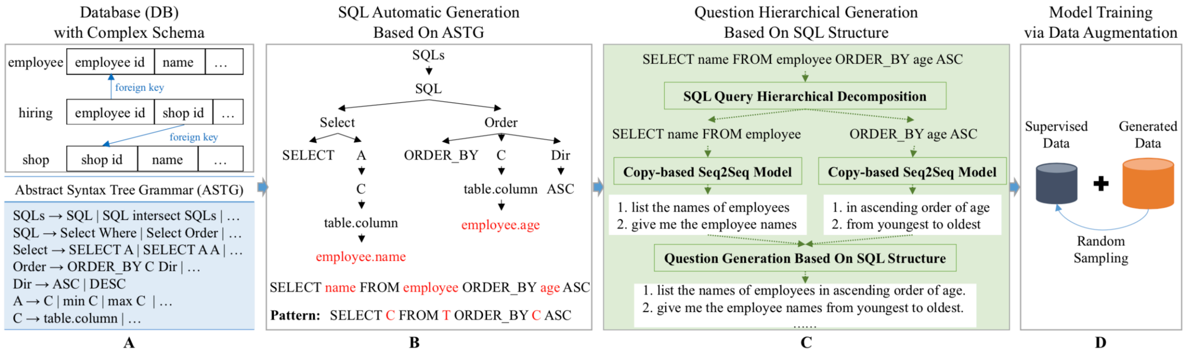

Database (DB)
with Complex Schema
employee
employee id
name
…
foreign key
hiring
employee id
shop id
…
foreign key
shop
shop id
name
…
Abstract Syntax Tree Grammar (ASTG)
SQLs → SQL | SQL intersect SQLs | …
SQL → Select Where | Select Order | …
Select → SELECT A | SELECT A A | …
Order → ORDER_BY C Dir | …
Dir → ASC | DESC
A → C | min C | max C | …
C → table.column | …
A
SQL Automatic Generation
Based On ASTG
SQLs
SQL
Select
Order
SELECT
A
ORDER_BY
C
Dir
C
table.column
ASC
table.column
employee.age
employee.name
SELECT name FROM employee ORDER_BY age ASC
Pattern: SELECT C FROM T ORDER_BY C ASC
B
Question Hierarchical Generation
Based On SQL Structure
SELECT name FROM employee ORDER_BY age ASC
SQL Query Hierarchical Decomposition
SELECT name FROM employee
ORDER_BY age ASC
Copy-based Seq2Seq Model
Copy-based Seq2Seq Model
1. list the names of employees
2. give me the employee names
1. in ascending order of age
2. from youngest to oldest
Question Generation Based On SQL Structure
1. list the names of employees in ascending order of age.
2. give me the employee names from youngest to oldest.
……
C
Model Training
via Data Augmentation
Supervised
Data
Generated
Data
Random
Sampling
D

Figure 2: An overview of our approach containing 3 stages: SQL query generation based on ASTG (Section §2.1), question hierarchical generation according to SQL structure (Section §2.2), model training via data augmentation (Section §2.4).

among questions and DB schemas, and achieve promising performance gains (Sun et al., 2018; Guo et al., 2019; Wang et al., 2020a).

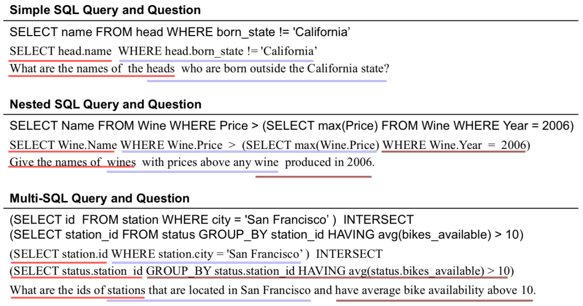
To handle the second challenge of lacking in labeled data, and inspired by the success of vanilla pretraining models with masked language model loss (Devlin et al., 2019), researchers propose task-specific pretraining models for semantic parsing (Yin et al., 2020; Herzig et al., 2020; Yu et al., 2020; Shi et al., 2020). The basic idea is to learn joint representations of structured data (i.e., tables) and corresponding contextual texts by designing delicate objective losses with large amounts of collected data that is related to the target task. These customized models have achieved good performance on English datasets. However, pretraining is slow and expensive as the models are trained on millions of web tables and related contexts. In addition, these approaches are currently only experimented on English since it is difficult to collect such data for pretraining.

This work follows another research line, i.e., data augmentation, which addresses both challenges discussed above in a resource-cheap way. The idea of data augmentation is automatically generating noisy labeled data using some deliberately designed method, and the technique has been successfully applied to a wide range of NLP tasks (Barzilay and McKeown, 2001; Jia and Liang, 2016). In our cross-domain text-to-SQL task, we can directly generate labeled data over unseen DBs as extra training data. The key of data augmentation is how to improve the quality of generated data. As two prior works, Yu et al. (2018a) manually align question tokens and DB elements in the corresponding SQL query, in order to obtain relatively high-quality question/SQL pairs, while Guo et al. (2018) utilize a flat Seq2Seq model to directly translate SQL queries to NL questions, which may only work for simple queries (see Section §4 for detailed discussion).

This work proposes a data augmentation framework with hierarchical SQL-to-question generation in order to obtain higher-quality question/SQL pairs. The framework consists of two steps. First, given a DB, we use an abstract syntax tree grammar (ASTG) to automatically generate SQL queries. For better distribution matching, we require the generated queries to cover at least 80% of SQL patterns in the original training data. Second, we design a hierarchical SQL-to-question generation model to obtain NL questions. The basic idea is: 1) decomposing a SQL query into clauses according to its syntax tree structure; 2) translating each clause into a subquestion; 3) concatenating subquestions into a full question according to the execution order of the SQL query. Finally, we design a simple sampling strategy to improve training efficiency with augmented data. In summary, we make the following contributions.

- We present a simple and resource-cheap data augmentation framework for cross-domain text-to-SQL parsing with no human intervention.[1]
- As the key component for our framework, we propose a hierarchical SQL-to-question generation model to obtain more reliable NL questions.
- In order to improve training efficiency, we propose a simple sampling strategy to utilize generated data, which is of relatively larger scale than original training data.

[1] We release the code at https://github.com/PaddlePaddle/Research/tree/master/NLP/Text2SQL-DA-HIER.



Simple SQL Query and Question

```
SELECT name FROM head WHERE born_state != 'California'
SELECT head.name  WHERE head.born_state != 'California'
What are the names of  the heads  who are born outside the California state?
```

Nested SQL Query and Question

```
SELECT Name FROM Wine WHERE Price > (SELECT max(Price) FROM Wine WHERE Year = 2006)
SELECT Wine.Name WHERE Wine.Price  >  (SELECT max(Wine.Price) WHERE Wine.Year  =  2006)
Give the names of  wines  with prices above any wine  produced in 2006.
```

Multi-SQL Query and Question

```
(SELECT id  FROM station WHERE city = 'San Francisco' )  INTERSECT
(SELECT station_id FROM status GROUP_BY station_id HAVING avg(bikes_available) > 10)
(SELECT station.id WHERE station.city = 'San Francisco' )  INTERSECT
(SELECT status.station_id GROUP_BY status.station_id HAVING avg(status.bikes_available) > 10)
What are the ids of stations that are located in San Francisco and have average bike availability above 10.
```

Figure 3: Examples of segment-level mapping between SQL queries and corresponding questions from the Spider dataset. In each example, the second SQL query is the equivalent of the first.

- We conduct experiments and analysis on three datasets in both English and Chinese, i.e., WikiSQL, Spider, and DuSQL, showing that our proposed framework can consistently improve performance over strong baselines.

2 Proposed Data Augmentation Approach

Given a DB, the goal of data augmentation is to automatically generate high-quality question/SQL pairs as extra training data. The key for its success lies in two aspects. First, the generated SQL queries should have similar distribution with the original data. Second, generated NL questions reflect the meaning of the corresponding SQL queries, especially for complex queries.

Our proposed framework adopts a two-step generation process, as shown in Figure 2. We first generate SQL queries at different complexity levels based on an ASTG, and then translate SQL queries into NL questions using our proposed hierarchical generation model.

2.1 SQL Query Generation

Being a program language, all SQL queries can be represented as nested tree structures, as depicted in Figure 2-B according to some context-free grammar. In fact, most text-to-SQL parsers proposed recently adopt the abstract syntax tree representation at the decoding stage (Yin and Neubig, 2018; Yu et al., 2018a; Guo et al., 2019; Wang et al., 2020a). Following those works, we design a general ASTG that can cover all SQL patterns in our adopted benchmark datasets. Due to space limitation, Figure 2 shows a fraction of the production rules.

According to our ASTG, the SQL query in Figure 2-B can be generated using the production rules: "*SQLs* $\rightarrow$ *SQL*", "*SQL* $\rightarrow$ *Select Order*", "*Select* $\rightarrow$ *SELECT A*", "*Order* $\rightarrow$ *ORDER_BY C Dir*", etc.

By assembling production rules from our ASTG, we can generate any *sketch tree*. As shown in Figure 2-B, a sketch tree means that DB-related leaf nodes (marked in red) are removed, and its flat form corresponds to *a pattern*, shown at the bottom. In our work, we generate sketch trees from simple to complex. Under a certain complexity level, i.e., tree breadth and depth, we first generate all possible sketch trees, and then apply them to new DBs to produce full trees (i.e., SQL queries) by filling DB-related items, such as table names, column names, and cell values.

In order to better match the query distribution of real text-to-SQL training data and to limit the number of generated SQL queries as well, we stop sketch tree generation when the generated ones cover more than 80% of patterns in the original training data[2]. The SQL queries are generated in a way that simpler SQL patterns come first, and 80% of the remaining patterns are usually high-frequency patterns. This limitation aims to control the complexity of generated questions, since very complex questions are rare in the training data. Please kindly note that our simple ASTG-based generation procedure can produce a lot of patterns unseen in the original data, because our generation is at production rule level. This is advantageous from the data variety perspective.

Moreover, given a DB, we only keep executable SQL queries for correctness check.

2.2 Hierarchical SQL-to-Question Generation

Given an SQL query, especially a complex one, it is difficult to generate an NL question that represents exactly same meaning. In their data augmentation work, Guo et al. (2018) use a vanilla Seq2Seq model to translate SQL queries into NL questions and obtain performance boost on WikiSQL consisting of simple queries. However, as shown in Table 2, we find performance consistently drops on all datasets over our strong baselines, which is largely due to the quality issue of generated NL questions, as illustrated in Table 3.

This work proposes a hierarchical SQL-to-

[2] As discussed in the logic form-based semantic parsing work of Herzig and Berant (2019), distribution mismatch is mainly caused by insufficient coverage of logical form templates.

question generation model to produce higher-quality NL questions. The idea is motivated by our observation that there is a strong segment-level mapping between SQL queries and corresponding questions, as shown in Figure 3. For example, the SQL query of the first example can be decomposed into two segments, i.e., the SELECT clause and the WHERE clause. The two clauses naturally correspond to the two question segments, i.e., "What are the names of the heads" and "heads who are born outside the California state" respectively.

Following the observation, our hierarchical SQL-to-question generation consists of three steps: 1) decomposing the given SQL query into several clauses; 2) translating every clause into a subquestion; 3) combining subquestions into a full NL question. Next we describe each step in detail.

Step 1: SQL clause decomposition. We decompose an SQL query into multiple clauses based on SQL keywords. Usually a clause contains only one keyword. In some cases multiple keywords are put into the same clause according to semantics. More formally speaking, multiple keywords are combined into one single clause based on two perspectives of consideration: 1) SQL syntax, and 2) alignment between SQL queries and NL questions, as illustrated by the last two examples in Figure 3.

From the perspective of SQL syntax, HAVING and GROUP_BY, are naturally bundled together, and thus are put into one clause, as shown in the third example of Figure 3. LIMIT and ORDER_BY are similarly handled.

From the second perspective, some keywords are not explicitly expressed in NL questions. In other words, there is a mismatch between intents expressed in NL questions and the implementation details in SQL queries. To better align them, we follow IRNet (Guo et al., 2019) and combine GROUP_BY with either SELECT or ORDER_BY.

For a nested SQL query, e.g., the second example in Figure 3, it is more reasonable to put the outside WHERE and the inside SELECT into one clause, since they together express a complete operation semantically.

Based on our decomposition method, an unseen SQL pattern always consists of common clause patterns in the training data.

Step 2: clause-to-subquestion translation. Compared with a full SQL query, a clause has a flat structure and involves simple semantics corresponding to a single SQL operation. Thus, it is much easier to translate clauses to subquestions compared with direct SQL-to-question translation. We use a standard copy-based Seq2Seq model (Gu et al., 2016) for clause-to-subquestion generation. The details are presented in Section §2.3.

Step 3: question composition. As shown in Figure 3, we compose a full question by concatenating all subquestions in a certain order. We experiment with two ordering strategies, i.e., the execution order of corresponding clauses[3], and the sequential order of corresponding clauses in the full SQL query. Preliminary experiments show that the former performs slightly better, which is thus adopted in our framework. Please note that direct concatenation may lead to redundant words from adjacent subquestions. We use several heuristic rules to handle this. Taking the third multi-SQL query in Figure 3 for example, since the two SELECT clauses are translated into nearly the same subquestion, we only keep one in the final NL question.

Discussion on quality of generated NL questions. Our reviewers suggest to evaluate naturalness and truth of generated NL questions. Due to time limitation, we did not perform strict manual evaluation. So far, our approach mainly considers the informativeness aspect of generated NL questions. We leave such evaluation and analysis as future work, which will certainly help us better understand our proposed approach.

2.3 Clause-to-subquestion Translation Model

We adopt the standard Seq2Seq model with copy mechanism (Gu et al., 2016) for clause-to-subquestion translation, which is also used in our baseline, i.e., flat SQL-to-question translation, with the same hyper-parameter settings.

In the input layer, we represent every SQL token by concatenating two embeddings, i.e., word (token as string) embedding, and token type (column/table/value/others) embedding, each having a dimension size of 150. We use default values for other hyper-parameters.

Training data construction. We construct clause/subquestion pairs for training the translation model from the original training data, consisting of two steps. The first step decomposes a SQL query

[3] The execution order used in our work is: WHERE -> GROUP BY -> HAVING -> SELECT, ORDER_BY -> LIMIT. For the SQL query with a nested part, we firstly generate the question for the nested part and use it as a fragment of the whole question, also known as the bottom-up way.

into clauses using the same way illustrated above.

The second step aims to decompose an NL question into clause-corresponding subquestions. In other words, this step finds a subquestion (i.e., a segment of the question) for each clause. We first build alignments between tokens in the SQL query and the corresponding NL question based on simple string matching. The string-matching method[4] is very similar to the schema linking step in IRNet (Guo et al., 2019) and RATSQL (Wang et al., 2020a). Then for each clause, we define the corresponding subquestion as the shortest question segment that contains all DB elements in the clause. Finally, we discard low-confidence clause/subquestion pairs to reduce noises, such as subquestions having large overlap with others. We keep overlapping subquestions, unless one subquestion fully contains another. In that case, we only keep the shorter subquestion.

We find that a portion of collected clauses have multiple subquestion translations. For example, the clause "*ORDER_BY age ASC*" are translated as both "in ascending order of the age" and "from youngest to oldest". We follow Hou et al. (2018) and use them as two independent clause/subquestion pairs for training.

2.4 Three Strategies for Utilizing Generated Data

Given a set of DBs, the generated question/SQL pairs are usually of larger scale than the original training data (see Table 1), which may greatly increase training time. In this work, we compare the following three strategies for parser training.

- **The pre-training strategy** first pre-trains the model with only generated data, and then fine-tunes the model with labeled training data.
- **The directly merging strategy** trains the model with all generated data and labeled training data in each epoch.
- **The sampling strategy** first randomly samples a number of generated data and trains the model on both sampled and labeled data in each epoch. The sampling size is set to be the same with the size of the labeled training data.

Dataset	Labeled Data	Generated Data
WikiSQL	61,297	98,206
Spider	8,625	58,691
DuSQL	18,602	45,942

Table 1: Data statistics in the number of question/SQL pairs. The column of labeled data shows the number of training pairs.

3 Experiments

Datasets. We adopt three widely-used cross-domain text-to-SQL parsing datasets to evaluate the effectiveness of different approaches, i.e., *WikiSQL*[5] and *Spider*[6] in English, and *DuSQL*[7] in Chinese. All datasets follow their original data splitting. WikiSQL focuses on single-table DBs and simple SQL queries that contain only one SELECT clause with one WHERE clause. In contrast, Spider and DuSQL are much more difficult in the sense each DB contains many tables and SQL queries may contain advanced operations such as clustering, sorting, calculation (only for DuSQL) and have nested or multi-SQL structures.

For each dataset, we generate a large number of question/SQL pairs against the evaluation DBs. Table 1 shows the size of the generated data. It is noteworthy that since the Spider test data is not publicly released, we generate data and evaluate different approaches against the Spider-dev DBs and question/SQL pairs.

Baseline parsers. We choose four popular open-source parsers to verify our proposed frameowrk.

WikiSQL: SQLova. The SQLova parser (Hwang et al., 2019) achieves competitive performance on WikiSQL without using execution guidance and extra knowledge (e.g., DB content and other datasets). The encoder obtains table-aware representations by applying BERT to concatenated sequence of question and table schema, and the decoder generates SQL queries as slot filling in the SELECT/WHERE clauses. HydraNet (Lyu et al., 2020) reported the state-of-the-art (SOTA) performance on WikiSQL but did not release their code.

Spider: IRNet and RATSQL. IRNet (Guo et al., 2019) is an efficient yet highly competitive parser for handling complex SQL queries on Spider, consisting of two novel components: 1) linking DB schemas with questions via string matching; 2)

[4] We extract all question n-grams ($1 \leq n \leq 6$) to match DB elements (i.e., columns, tables, and values) in the corresponding SQL query, so as to get alignments between question n-grams and DB elements.

[5] https://github.com/salesforce/WikiSQL

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

[7] https://aistudio.baidu.com/aistudio/competition/detail/47

a grammar-based decoder to generate SemQL trees as intermediate representations of SQLs. RATSQL (Wang et al., 2020a) is the current SOTA parser on Spider. The key contribution is utilizing a relation-aware transformer encoder to better model the connections between DB schemas and NL questions. However, training RATSQL is very expensive. It takes about 7 days to train a basic BERT-enhanced RATSQL model on a V100 GPU card, which is about 10 times slower than IRNet.

With limited computational resource, we mainly use IRNet for model ablation and efficiency comparison. Meanwhile, we report main results on RATSQL to learn the effect of our proposed framework on more powerful parsers.

Another detail about RATSQL to be noticed is that we use the released Version 2 (V2). They reported higher performance with V3[8] by better hyper-parameter settings and even longer training time. However, they did not release their configurations.

DuSQL: IRNet-Ext. IRNet-Ext proposed by Wang et al. (2020b) is an extended version of IRNet to accommodate the characteristics of Chinese dataset DuSQL. In this work, we further enhance IRNet-Ext with BERT. Basically, we concatenate the NL question and DB schema as the input, and perform encoding with BERT (instead of BiLSTM in the original parser).

Evaluation metrics. We use the exact matching (EM) accuracy as the main metric, meaning the percentage of questions whose predicted SQL query is equivalent to the gold SQL query, regardless of clause and component ordering. We also use component matching (CM) F1 score to evaluate the clause-level performance for in-depth analysis. Besides, we report execution (exec) accuracy on WikiSQL, meaning the percentage of questions whose predicted SQL query obtains the correct answer.

Hyper-parameter settings. For each parser, we use default parameter settings in their released code. All these parsers are enhanced with vanilla (in contrast to task-specific) pretraining models, i.e., BERT (Devlin et al., 2019), including IRNet-Ext.

In order to avoid the effect of performance vibrations[9], we run each model for 5 times with different random initialization seeds, and report the averaged EM accuracy (mean) and the variance ($\sqrt{\frac{\sum_{i=1}^{n}(x_i-\bar{x})^2}{n-1}}$). We only run each RATSQL model for 3 times due to its prohibitively high requirement on computational resource.

WikiSQL	
Models	EM [Exec]
HydraNet (Lyu2020)	83.8 [89.2]
SQLova (Hwang2019)	80.7 [86.2]
STAMP (Guo2018)	60.7 [74.4]
+ Aug (FLAT)	63.7 (+3.0) [75.5]
SQLova (ours)	80.1$_{\pm 0.40}$ [85.7]
+ Aug (FLAT)	79.7$_{\pm 0.50}$ (−0.4) [85.4]
+ Aug (HIER)	81.2$_{\pm 0.09}$ (+1.1) [86.5]
Spider	
Models	EM
IRNet (Guo2019)	60.6
RATSQL V3 (Wang2020a)	69.6
RATSQL V2 (Wang2020a)	65.8
SyntaxSQLNet (Yu2018a)	22.1
+ Aug (PATTERN)	28.7 (+6.6)
IRNet (ours)	59.7$_{\pm 0.41}$
+ Aug (FLAT)	58.8$_{\pm 0.56}$ (−0.9)
+ Aug (HIER)	61.8$_{\pm 0.32}$ (+2.1)
RATSQL V2 (ours)	65.4$_{\pm 0.60}$
+ Aug (HIER)	68.2$_{\pm 0.42}$ (+2.8)
DuSQL	
Models	EM
IRNet-Ext (Wang2020b)	50.1
IRNet-Ext + BERT (ours)	53.7$_{\pm 0.60}$
+ Aug (FLAT)	53.4$_{\pm 0.67}$ (−0.3)
+ Aug (HIER)	60.5$_{\pm 0.40}$ (+6.8)

Table 2: Main results. We run each model for 5 times, and report the average and variance (as subscripts). +Aug means the model is enhanced with data augmentation, and *PATTERN*, *FLAT*, and *HIER* refer to the three data augmentation approaches.

3.1 Main Results

Table 2 shows the main results. For each dataset, the first major row shows previously reported results, and the second major row gives results of our base parsers without and with data augmentation.

To compare previous data augmentation methods, we also re-implement the flat one-stage generation approach (FLAT) proposed by Guo et al. (2018). We do not implement the pattern-based data augmentation approach (PATTERN) of Yu et al. (2018a) due to its requirement of human intervention. Moreover, their large performance improvement is obtained over a very weak baseline.

Performance of our baseline parsers. On WikiSQL, the averaged performance of our SQLova parser is lower than their reported performance by about 0.7. On Spider, the performance of our IR-

[8] The comparison of V2 and V3 is discussed at https://github.com/microsoft/rat-sql/issues/12.

[9] Please see issues proposed at the github of RATSQL model, such as https://github.com/microsoft/rat-sql/issues/10.

SQL	SELECT draw_size FROM matches
	WHERE loser_age > 10

FLAT	what are the percentage of draw size in matches with loser higher than 10?
HIER	with losers who are older than 10, find the draw size of the matches.

SQL	SELECT horsepower FROM cars_data
	WHERE edispl <= 10
	ORDER_BY year DESC

FLAT	list all horsepower year in descending order of year.
HIER	with edispl no higher than 10, show the horsepower of the cars, made from most recently to oldest.

Table 3: Case study on FLAT vs. HIER on a simple SQL query (first) and a complex one (second) from Spider.

Net parser is lower than their reported value by 0.9. However, please kindly note that we use default configurations of SQLova and IRNet, and our best results among five runs on both WikiSQL and Spider are very close to theirs.

As discussed earlier, HydraNet and RATSQL v3 achieve higher performance, but they do not release their code or configurations.

In summary, we can conclude that our baseline parsers achieve competitive results on all three datasets. We believe that it would be more reasonable to report the mean and variance of performance.

Comparison of different data augmentation methods. According to our results in the second major row of each dataset, data augmentation with FLAT leads to consistent performance degradation, which is contradictory to the results on WikiSQL reported by Guo et al. (2018). We suspect the reason is that our BERT-enhanced baseline parser is much stronger than their adopted parser. To verify this, we run SQLova without BERT and find similar performance gains from 61.0% to 64.0% via the FLAT data augmentation. Using HIER, the performance can further increase to 66.1%. Due to time and resource limitation, we do not run similar experiments on the other two datasets.

In contrast to FLAT, our proposed HIER approach achieves consistent improvement over the strong BERT-enhanced parsers. In particular, it is very interesting to see that the parsers have lower performance variance compared with the baselines. We will give more insights on the effectiveness of the hierarchical generation approach in Section §3.2. Again, to save computational resource, we did not run RATSQL with the FLAT data augmentation approach.

Looking closer into the improvements on the three datasets, we can see that our HIER data augmentation obtains the least performance increase on WikiSQL, possibly due to the higher baseline performance with relatively large-scale labeled data consisting of simple SQL queries. The most gain is obtained on DuSQL. We suspect the reason is twofold. First, the baseline performance is the lowest, which is similar to the results obtained by Yu et al. (2018a) on Spider with data augmentation. Second, during the construction of DuSQL, Wang et al. (2020b) first automatically generate question/SQL pairs and then perform manual correction and paraphrasing, leading to certain resemblance between their labeled data and our generated data.

In summary, we can conclude that our proposed augmentation approach with hierarchical SQL-to-question generation is more effective than previous methods, and can substantially improve performance over strong baselines, especially over complex datasets. In the future, we would like to apply our approach to other text-to-SQL datasets and languages.

3.2 Analysis

Case study. To intuitively understand the advantages of HIER over FLAT, we present two typical examples in Table 3. The FLAT approach fails to understand the column name "loser_age" in the WHERE clause of the first SQL query, and overlooks the WHERE clause completely in the second query. In contrast, our HIER approach basically captures semantics of both two SQL queries, though the generated questions seem a little bit unnatural due to the ordering issue. Under our hierarchical generation approach, clause-to-subquestion translation is much simpler than direct SQL-to-question translation, hence leading to relatively high-quality NL questions.

Component-level analysis. To understand fine-grained impact of our proposed augmentation framework, we report CM F1 scores over five types of SQL clauses in Table 4. We observe that the main advantage of data augmentation on WikiSQL comes from the prediction of WHERE clause, which is also the main challenge of simple datasets. The performance of SELECT clause is near the upper bound, where most of the evaluation errors are due to wrong annotations by humans (Hwang et al., 2019). For Spider, performances of all clauses are

Datasets	Models	SELECT	WHERE	GROUP_BY	HAVING	ORDER_BY
WikiSQL	SQLova	88.1	90.2	–	–	–
	+ Aug	87.8 (−0.3)	91.6 (+1.4)	–	–	–
Spider	IRNet	87.7	68.0	80.8	75.5	76.3
	+ Aug	88.5 (+0.8)	70.1 (+2.1)	81.0 (+0.2)	77.2 (+1.7)	80.0 (+4.5)
	RATSQL	85.5	72.6	79.3	76.7	79.4
	+ Aug	87.7 (+2.2)	75.4 (+2.8)	82.5 (+3.2)	79.4 (+2.7)	80.9 (+1.5)
DuSQL	IRNet-Ext	78.5	82.4	93.8	92.1	93.4
	+ Aug	79.8 (+1.3)	86.6 (+4.2)	95.0 (+1.2)	93.3 (+1.1)	93.5 (+0.1)

Table 4: CM F1 scores over five types of SQL clauses. The type division is borrowed from Yu et al.(2018b).

Models	Seen patterns	Unseen patterns
IRNet	63.5	48.8
IRNet + Aug	64.7 (+1.2)	53.7 (+4.9)
RATSQL	66.6	52.3
RATSQL + Aug	73.0 (+6.4)	55.4 (+3.1)

Table 5: EM accuracy over seen and unseen patterns on Spider.

improved. Looking into the Spider dataset, we find that our generated subquestions are of high quality in the terms of diversity and semantics, e.g., "age" translated as "from youngest to oldest", and "year" as "recent". It is interesting to see that performances of the right-side three types of complex clauses are much higher on DuSQL than on Spider, and also much higher than that of the basic SELECT/WHERE clauses on DuSQL itself. As discussed earlier in Section §3.1, we suspect this is because the complex clauses on DuSQL are more regularly distributed and thus more predictable due to their data construction method.

Analysis on SQL patterns. One potential advantage of ASTG-based SQL generation is the ability to generate new SQL patterns that do not appear in the training data. To verify this, we adopt the more complex Spider, since its evaluation data contains a lot (20%) of low-frequency SQL patterns unseen in the training data. We divide the question/SQL pairs into two categories according to the corresponding SQL pattern, and report EM accuracy in Table 5. It is clear that our augmentation approach gains improvement both on seen and unseen patterns. The gains on unseen patterns show that with generated data as extra training data, the model possesses better generalization ability.

Impact of augmented data size. We study how the number of augmented pairs affects the accuracy of parsing models. We conduct this experiment on the Spider dataset using IRNet model based on the directly merging training strategy. In the experiment, we randomly sample question/SQL pairs from all the generated data based on multiples of the size of the original training data. Results are given in Table 6. It is not surprising that more augmented data brings higher accuracy which is consistent with the observations in Guo et al. (2018). Interestingly, we find that more augmented data brings more stable benefits.

Size	100%	200%	300%	all
Acc	59.1 (±1.18)	59.4 (±0.75)	59.3 (±0.69)	61.8 (±0.26)

Table 6: The impact of augmented data size on Spider using IRNet model. The numbers in brackets represent the variance of three runs.

Strategies	EM Accuracy	Total Training Time
Baseline	59.5	6.9 hours
Pre-training	60.0	36.1 hours
Directly merging	61.8	34.9 hours
Sampling	61.7	10.4 hours

Table 7: Comparison of three training strategies on Spider using IRNet model. The size of augmented data is about 6.7 times that of the original training data, as shown in Table 1.

Comparison on training strategies. Table 7 compares the three training strategies for utilizing generated data, which are discussed in Section §2.4. All experiments are run on one V100 GPU card. The pre-training strategy only slightly improves performance over the baseline, indicating that it fails to make full use of the generated data. The directly merging strategy and the sampling strategy achieve nearly the same large improvement. However, the sampling strategy is much more efficient.

4 Related Work

Data augmentation for NLP. As an effective way to address the sparseness of labeled data, data augmentation has been widely and successfully adopted in the computer vision field (Szegedy et al., 2015). Similarly in the NLP field, a wide range of tasks employ data augmentation to accommodate the capability and need of deep learning models in consuming big data, e.g., text-classification (Wei and Zou, 2019), low-resource dependency parsing (Şahin and Steedman, 2018), machine translation (Fadaee et al., 2017), etc. Concretely, the first kind of typical techniques tries to generate new data by manipulating the original instance via word/phrase replacement (Wang and Yang, 2015; Jia and Liang, 2016), random deletion (Wei and Zou, 2019), or position swap (Şahin and Steedman, 2018; Fadaee et al., 2017). The second kind creates completely new instances via generative models (Yoo et al., 2019), while the third kind uses heuristic patterns to construct new instances (Yu et al., 2018a).

Data augmentation for semantic parsing. Given an NL question and a knowledge base, semantic parsing aims to generate a semantically equivalent formal representation, such as SQL query, logic form (LF), or task-oriented dialogue slots. Based on LF-based representation, Jia and Liang (2016) train a synchronous context free grammar model on labeled data for generating new question/LF pairs simultaneously. Hou et al. (2018) focus on the slot filling task. They train a Seq2Seq model on semantically similar utterance pairs, and generate new and diverse utterances for each original one.

Data augmentation for text-to-SQL parsing. Iyer et al. (2017) focus on in-domain text-to-SQL parsing. They automatically translate NL questions into SQL queries, and ask human experts to correct unreliable SQL queries. On Spider, Yu et al. (2018a) collect many high-frequency SQL patterns and also convert corresponding questions into patterns by removing the concrete database-related tokens. They keep 50 high-quality question/SQL pattern pairs via manual check, and use them to generate new question/SQL pairs for a given table.[10] However, their approach only considers SQL patterns concerning single table, and the need for human intervention seems expensive. Guo et al. (2018) use a pattern-based approach to generate SQL queries and utilize a copy-based Seq2Seq model to directly translate SQL queries into NL questions. In contrast, this work proposes to use an ASTG for better SQL query generation and a hierarchical SQL-to-question generation approach to obtain higher-quality NL questions.

[10] More specifically, given a pair of question and SQL query, they first manually align question tokens and DB elements, and replace the aligned terms with some special, generic symbols, resulting in question/SQL templates. Then given a new table, they generate question/SQL pairs by filling question/SQL templates with table elements.

5 Conclusions

This paper presents a simple yet effective automatic data augmentation framework for cross-domain text-to-SQL parsing. With two-step processing, i.e., ASTG-based SQL query generation and hierarchical SQL-to-question generation, our framework is able to produce high-quality question/SQL pairs on given DBs. Results on three widely used datasets, i.e., WikiSQL, Spider, and DuSQL show that: 1) the hierarchical generation component is the key for performance boost, due to the more reliable clause-to-subquestion translation, and in contrast, previously proposed direct SQL-to-question generation leads to performance drop over strong baselines; 2) our proposed framework can consistently boost performance on different types of SQL clauses and patterns; 3) the sampling strategy is superior to the other two strategies for training parsers with both labeled and generated data, especially in the terms of training efficiency.

Acknowledgments

We are very grateful to our anonymous reviewers for their helpful feedback on this work. Kun Wu, Zhenghua Li and Min Zhang were supported by National Natural Science Foundation of China (Grant No. 62176173 and 61876116), and a Project Funded by the Priority Academic Program Development (PAPD) of Jiangsu Higher Education Institutions.

References

Regina Barzilay and Kathleen McKeown. 2001. Extracting paraphrases from a parallel corpus. In *ACL*, pages 50–57.

C... J. Date and Hugh Darwen. 1997. *A Guide to the SQL Standard: A user's guide to the standard database language SQL*. Addison-Wesley.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*, pages 4171–4186.

Marzieh Fadaee, Arianna Bisazza, and Christof Monz. 2017. Data augmentation for low-resource neural machine translation. In *ACL*, pages 567–573.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor OK Li. 2016. Incorporating copying mechanism in sequence-to-sequence learning. In *ACL*.

Daya Guo, Yibo Sun, Duyu Tang, Nan Duan, Jian Yin, Hong Chi, James Cao, Peng Chen, and Ming Zhou. 2018. Question generation from sql queries improves neural semantic parsing. In *EMNLP*.

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

Jonathan Herzig and Jonathan Berant. 2019. Don't paraphrase, detect! rapid and effective data collection for semantic parsing. In *EMNLP-IJCNLP*, pages 3801–3811.

Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Eisenschlos. 2020. TaPas: Weakly supervised table parsing via pre-training. In *ACL*, pages 4320–4333.

Yutai Hou, Yijia Liu, Wanxiang Che, and Ting Liu. 2018. Sequence-to-sequence data augmentation for dialogue language understanding. In *COLING*.

Wonseok Hwang, Jinyeong Yim, Seunghyun Park, and Minjoon Seo. 2019. A comprehensive exploration on wikisql with table-aware word contextualization. *arXiv:1902.01069*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. In *ACL*, pages 963–973.

Robin Jia and Percy Liang. 2016. Data recombination for neural semantic parsing. In *ACL*.

Fei Li and HV Jagadish. 2014. Constructing an interactive natural language interface for relational databases. *VLDB Endowment*, 8(1).

Qin Lyu, Kaushik Chakrabarti, Shobhit Hathi, Souvik Kundu, Jianwen Zhang, and Zheng Chen. 2020. Hybrid ranking network for text-to-sql. *arXiv preprint arXiv:2008.04759*.

Gözde Gül Şahin and Mark Steedman. 2018. Data augmentation via dependency tree morphing for low-resource languages. In *EMNLP*.

Peng Shi, Patrick Ng, Zhiguo Wang, Henghui Zhu, Alexander Hanbo Li, Jun Wang, Cicero Nogueira dos Santos, and Bing Xiang. 2020. Learning contextual representations for semantic parsing with generation-augmented pre-training. *arXiv preprint arXiv:2012.10309*.

Yibo Sun, Duyu Tang, Nan Duan, Jianshu Ji, Guihong Cao, Xiaocheng Feng, Bing Qin, Ting Liu, and Ming Zhou. 2018. Semantic parsing with syntax-and table-aware sql generation. In *ACL*, pages 361–372.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020a. RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. In *ACL*, pages 7567–7578.

Lijie Wang, Ao Zhang, Kun Wu, Ke Sun, Zhenghua Li, Hua Wu, Min Zhang, and Haifeng Wang. 2020b. DuSQL: A large-scale and pragmatic Chinese text-to-SQL dataset. In *EMNLP*.

William Yang Wang and Diyi Yang. 2015. That's so annoying!!!: A lexical and frame-semantic embedding based data augmentation approach to automatic categorization of annoying behaviors using# petpeeve tweets. In *EMNLP*, pages 2557–2563.

Jason Wei and Kai Zou. 2019. Eda: Easy data augmentation techniques for boosting performance on text classification tasks. *arXiv:1901.11196*.

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. Sqlizer: query synthesis from natural language. *ACM*, 1(OOPSLA):1–26.

Pengcheng Yin and Graham Neubig. 2018. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *EMNLP*, pages 7–12.

Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. TaBERT: Pretraining for joint understanding of textual and tabular data. In *ACL*.

Kang Min Yoo, Youhyun Shin, and Sang-goo Lee. 2019. Data augmentation for spoken language understanding via joint variational generation. In *AAAI*.

Tao Yu, Chien-Sheng Wu, Xi Victoria Lin, Bailin Wang, Yi Chern Tan, Xinyi Yang, Dragomir Radev, Richard Socher, and Caiming Xiong. 2020. Grappa: Grammar-augmented pre-training for table semantic parsing. *arXiv:2009.13845*.

Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. 2018a. Syntaxsqlnet: Syntax tree networks for complex and cross-domain text-to-sql task. In *EMNLP*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018b. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *EMNLP*, pages 3911–3921.

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