Measuring and Improving Compositional Generalization in Text-to-SQL via Component Alignment

Yujian Gan1  Xinyun Chen2  Qiuping Huang4  Matthew Purver1,3
1Queen Mary University of London  2UC Berkeley  3Jožef Stefan Institute
4Nanning Central Sub-branch of the People’s Bank of China

{y.gan,m.purver}@qmul.ac.uk  xinyun.chen@berkeley.edu
qiuping_h@foxmail.com

Abstract

In text-to-SQL tasks — as in much of NLP — compositional generalization is a major challenge: neural networks struggle with compositional generalization where training and test distributions differ. However, most recent attempts to improve this are based on word-level synthetic data or specific dataset splits to generate compositional biases. In this work, 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.1

1 Introduction

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). This 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. (2021) 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., 2020; Wang et al., 2021; Liu et al., 2020) improve generalization by enhancing the model’s component awareness. Similarly, 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 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 work, we first introduce a new dataset, Spider-SS (SS stands for sub-sentence), derived from Spider (Yu et al., 2018b); Figure 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 cor-

---

1Our code and dataset are available at https://github.com/ygan/SpiderSS-SpiderCG
Figure 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.

response SQL clause, reducing the difficulty of this task by using the intermediate representation language NatSQL (Gan et al., 2021b), 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.

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 2 shows one example for each. The first subset contains 23,569 examples generated by substituting sub-sentences; we consider most data in this subset as in-distribution. The second subset contains 22,030 examples generated by appending sub-sentences, increasing the length and complexity of the sentence and the SQL query compared to the original samples; we consider this subset as 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.

In short, we make the following contributions:

- 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.

Figure 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.

2 Spider-SS

2.1 Overview

Figure 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., 2021b) 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.

2.2 Sentence Split Algorithm

We build our sentence split algorithm upon the NL dependency parser spaCy 2, 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 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 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.

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 4, the last two sub-sentence should be combined to correspond to “ORDER BY Customer.Email_Address, Customer.Phone_Number 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.
2.3 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 4, the “in ascending order” sub-sentence does not mention the “Farm.Total_Horses” column. Therefore, we add a “NO MENTIONED” 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 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 5, we can use NatSQL to complete the annotation quickly, while the NatSQL can be converted back to the target SQL. The detail of the annotation steps can be found in Appendix C.

3 Spider-CG

3.1 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 named the Spider-CG subset generated by the sub-sentence substitution method CG-SUB, and the other named CG-APP.

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.

3.2 Generation Algorithm

According to Algorithm 1, 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 1 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 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 5. In contrast, no element is extracted from the example in Figure 1. It should be noted that elements in a do
Algorithm 1 Generate CG-SUB and CG-APP dataset in a certain domain

Input: $e_{\text{list}}$ ⊿ 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 $e_{\text{list}}$ do
2:   for Every $element_2$ in $e_{\text{list}}$ do
3:     if $element_1 \neq element_2$ then
4:       if $element_1$.can_be_substituted_by($element_2$) then
5:         $cg_{\text{sub}}$.append($element_1$.generate_substitution_example($element_2$))
6:       if $element_1$.can_append($element_2$) then
7:         $cg_{\text{app}}$.append($element_1$.generate_appending_example($element_2$))
8: return $cg_{\text{sub}}, cg_{\text{app}}$

Ques Show the name of employees named Mark Young?

SQL

```
SELECT name FROM employee
WHERE name = 'Mark Young'
```

Table 1: One acceptable but not perfect examples in the Spider-CG.

main cannot be used in another because the schema items are different. So as many domains as there are, it needs to run Algorithm 1 as many times. We recommend reading Appendix A for details of can_be_substituted_by and can_append functions.

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

4 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 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 6 presents the workflow of encoding the whole sentence but decoding only the sub-sentence.
The 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.

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 6 is split to two examples shown in Figure 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 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.

5 Experiment

5.1 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 3.1, we divide the Spider-CG into two subsets: CG-SUB and CG-APP. Therefore, there are five evaluation sets:

- **SpiderD**: the original Spider development set with 1,034 examples for cross-domain in-distribution text-to-SQL evaluation.
- **CG-SUBT**: the CG-SUB training set, containing 20,686 examples generated from Spider-SS training set by substituting sub-sentences. CG-SUBT can be used for in-domain in-distribution text-to-SQL evaluation.
- **CG-SUBD**: the CG-SUB development set containing 2,883 examples for cross-domain in-distribution text-to-SQL evaluation.
- **CG-APPT**: the CG-APP training set, containing 18,793 examples generated from Spider-SS training set by appending sub-sentences. CG-APPT can be used for in-domain out-of-distribution text-to-SQL evaluation.
- **CG-APPD**: 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., 2019) model using the GLOVE (Pennington et al., 2014) embeddings.
- **RATSQL**: The RATSQL (Wang et al., 2020) model using the GLOVE embeddings.
- **RATSQLB**: The RATSQL model using the BERT (Devlin et al., 2019) embeddings.
- **RATSQLG**: The RATSQL model using the GAP (Shi et al., 2021) embeddings.
- **(N)**: This subscript indicates that the model use NatSQL instead of SQL.
- **(S)**: This subscript indicates that the model is modified according to Section 4 and trained on Spider-SS. Besides, since Spider-SS is annotated

3Out-of-distribution means that the difficulty distribution is different from the Spider; see Table 3. Appendix A discusses the removal of overly complex examples to ensure that Spider-CG’s SQL does not exceed the complexity upper bound of the Spider.
Table 2: 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.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Exact Match</th>
<th>Execution Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training</td>
<td>90.7%</td>
<td>93.3%</td>
</tr>
<tr>
<td>Development</td>
<td>94.8%</td>
<td>95.2%</td>
</tr>
</tbody>
</table>

Table 3: The difficulty distribution of five different evaluation sets.

<table>
<thead>
<tr>
<th></th>
<th>easy</th>
<th>medium</th>
<th>hard</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spider_D</td>
<td>24.1%</td>
<td>43.1%</td>
<td>16.8%</td>
<td>16.1%</td>
</tr>
<tr>
<td>CG-SUB_T</td>
<td>28.6%</td>
<td>38.0%</td>
<td>21.1%</td>
<td>12.3%</td>
</tr>
<tr>
<td>CG-SUB_D</td>
<td>37.6%</td>
<td>38.4%</td>
<td>12.0%</td>
<td>12.0%</td>
</tr>
<tr>
<td>CG-APP_T</td>
<td>3.3%</td>
<td>31.4%</td>
<td>26.0%</td>
<td>39.3%</td>
</tr>
<tr>
<td>CG-APP_D</td>
<td>2.4%</td>
<td>44.3%</td>
<td>22.9%</td>
<td>30.4%</td>
</tr>
</tbody>
</table>

by NatSQL, this subscript also indicates that the model uses NatSQL instead of SQL.

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\_B(S). RATSQL\_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.

5.2 Dataset Analysis

Spider-SS. Table 2 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., 2021b) because the Spider-SS uses equivalent SQL and corrects some errors, as discussed in Section 2.3. 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\_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., 2021b). Thus, we recommend using NatSQL-based datasets to evaluate models trained on NatSQL.

Spider-CG. Table 3 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 et al., 2021; Gan et al., 2021b). It can be found from Table 3 that the difficulty distribution of CG-SUB\_T and CG-SUB\_D is similar to that of Spider\_D. The similar distributions among CG-SUB\_T, CG-SUB\_D, and Spider\_D support the view discussed in Section 1 that the examples generated by the substitution method are in-distribution.

On the other hand, the difficulty distributions of CG-APP\_T and CG-APP\_D are obviously different from that of Spider\_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.

5.3 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 2.2. 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 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 4 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 2.2, 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,
Table 4: 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.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Deviation &lt;= 1</th>
<th>Deviation &lt;= 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG-SUBₜ</td>
<td>93.2%</td>
<td>94.4%</td>
</tr>
<tr>
<td>CG-SUBₜ</td>
<td>92.9%</td>
<td>94.1%</td>
</tr>
<tr>
<td>CG-APPₜ</td>
<td>86.0%</td>
<td>90.4%</td>
</tr>
<tr>
<td>CG-APPₜ</td>
<td>88.9%</td>
<td>92.6%</td>
</tr>
</tbody>
</table>

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.

5.4 Model Results

Table 5 presents the exact match accuracy on the five different evaluation sets. In the two OOD datasets, CG-APPₜ and CG-APPₜ, 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 RATSQL₆(N) 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 RATSQL₆(S) are not workable, and the performance of GNN on the SpiderD and CG-SUBD is degraded.

Each model has a close result between the unseen SpiderD and CG-SUBD, indicating that from the perspective of the model, the synthetic sentences are pretty similar to NL. Therefore, we believe the performance on CG-SUBD can be generalized to the real world. Moreover, considering that the algorithms for generating CG-SUBD and CG-APPD are close (see Appendix A), we can further speculate that the synthetic sentences of CG-APPD 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., 2021b) show that NatSQL improve the model performance in extra hard SQL. Therefore, RATSQL₆(G₆(N)) and RATSQL₆(B₆(N)) suffer less performance degradation in CG-APPₜ and CG-APPₜ than RATSQL₆(G₆) and RATSQL₆(B₆).

6 Limitation of this Work

The Spider-SS and Spider-CG are based on Spider, an English large-scale text-to-SQL dataset, and we did not extend the experiment to other language and text-to-SQL datasets. Therefore, we did not verify whether these methods work well in other languages and datasets. Besides, since this work is based on NatSQL, there will be around 5% of NatSQL that can not be converted to the correct SQL.

7 Related Work

Data augmentation for text-to-SQL models. Data augmentation has been commonly used for improving performance (Xiong and Sun, 2019; Li et al., 2019). In the context of text-to-SQL generation, Yu et al. (2018a) generate synthetic training samples from some pre-defined SQL and NL question templates. Parikh et al. (2020) introduces a table-to-text dataset with over 120,000 examples that proposes a controlled generation task: given a Wikipedia table and a set of highlighted table cells, produce a one-sentence description. Yu et al. (2021) sample from the given examples and then give a large number of tables to generate new synthetic examples. Shi et al. (2021) present a model pre-training framework that jointly learns representations of NL utterances and table schemas by leveraging generation models to generate pre-train data. Our proposed Spider-CG dataset can be used for data augmentation.

Compositional generalization for semantic parsing. Compositional generalization for semantic parsing has captured wide attention recently (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
Table 5: Exact match accuracy on evaluation sets.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Spider\textsubscript{D}</th>
<th>CG-SUB\textsubscript{T}</th>
<th>CG-SUB\textsubscript{D}</th>
<th>CG-APP\textsubscript{T}</th>
<th>CG-APP\textsubscript{D}</th>
</tr>
</thead>
<tbody>
<tr>
<td>RATSQL\textsubscript{G}</td>
<td>72.7%</td>
<td>80.9%</td>
<td>70.3%</td>
<td>45.2%</td>
<td>44.2%</td>
</tr>
<tr>
<td>RATSQL\textsubscript{G(N)}</td>
<td>73.9%</td>
<td>90.2%</td>
<td>75.0%</td>
<td>67.8%</td>
<td>60.5%</td>
</tr>
<tr>
<td>RATSQL\textsubscript{G(S)}</td>
<td>74.5%</td>
<td>91.4%</td>
<td>76.7%</td>
<td>82.5%</td>
<td>68.3%</td>
</tr>
<tr>
<td>RATSQL\textsubscript{B}</td>
<td>72.0%</td>
<td>79.5%</td>
<td>72.0%</td>
<td>45.1%</td>
<td>47.2%</td>
</tr>
<tr>
<td>RATSQL\textsubscript{B(N)}</td>
<td>72.1%</td>
<td>83.2%</td>
<td>69.4%</td>
<td>54.6%</td>
<td>53.1%</td>
</tr>
<tr>
<td>RATSQL\textsubscript{B(S)}</td>
<td>71.9%</td>
<td>91.0%</td>
<td>72.6%</td>
<td>79.8%</td>
<td>61.5%</td>
</tr>
<tr>
<td>RATSQL\textsubscript{(N)}</td>
<td>63.2%</td>
<td>79.1%</td>
<td>60.7%</td>
<td>40.6%</td>
<td>34.5%</td>
</tr>
<tr>
<td>RATSQL\textsubscript{(S)}</td>
<td>64.7%</td>
<td>88.8%</td>
<td>63.3%</td>
<td>72.1%</td>
<td>44.1%</td>
</tr>
<tr>
<td>GNN\textsubscript{(N)}</td>
<td>54.4%</td>
<td>67.3%</td>
<td>57.5%</td>
<td>30.4%</td>
<td>25.1%</td>
</tr>
<tr>
<td>GNN\textsubscript{(S)}</td>
<td>49.3%</td>
<td>71.9%</td>
<td>51.8%</td>
<td>52.1%</td>
<td>34.6%</td>
</tr>
</tbody>
</table>

new database schemas (Suhr et al., 2020; Gan et al., 2021a). On the other hand, Shaw et al. (2021) introduces TMCD splits for studying compositional generalization in semantic parsing, where they aim to maximize the divergence of SQL compounds between the training and test sets.

Although both the TMCD split and our Spider-CG can be used to evaluate the text-to-SQL compositional generalization ability, their problem setting is different. TMCD split is based on SQL syntax structure, while Spider-CG is based on the natural language syntax, which leads to different requirements for compositional generalization ability. For example, TMCD splits require model learning “Give me the name of students who is the oldest” can predict the “Give me the name of the oldest student” since their SQL is the same. Spider-CG does not expect the model to do so because the syntax of questions is different, i.e., “Give me the name of students who is the oldest” contains two sub-sentences, and none of them is close to the “Give me the name of the oldest student”. In other words, Spider-CG requires the model learning “List the id of the oldest dog” can predict the “Give me the name of the oldest student”.

Our model is inspired by prior works on neural parsers constructed to capture granular information from a whole. Yin et al. (2021) describe a span-level supervised attention loss that improves compositional generalization in semantic parsers. Herzig and Berant (2021) propose SpanBasedSP, a parser that predicts a span tree over an input utterance, and dramatically improves performance on splits that require compositional generalization. Chen et al. (2020) propose the Neural-Symbolic Stack machine (NeSS), which integrates a symbolic stack machine into a seq2seq generation framework, and learns a neural network as the controller to operate the machine. 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. Our proposed Spider-SS can be used to replace or evaluate the alignment algorithm.

8 Conclusion

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.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. Matthew Purver acknowledges financial support from the UK EPSRC under grant EP/S033564/1, and from the Slovenian Research Agency for research core funding (No. P2-0103 and No. P5-0161). Xinyun Chen is supported by the Facebook Fellowship.
References


Divyansh Kaushik, Eduard Hovy, and Zachary Lipton. 2020. Learning the difference that makes a difference with counterfactually-augmented data. In International Conference on Learning Representations.


Logic checks are used to prevent generating contradictory examples. First, logic checks filter out examples that are not consistent with the input data. This includes checking for contradictions in the generated sentences, such as incorrect conditionals or logical inconsistencies. Logic checks are particularly important when dealing with complex and cross-domain text-to-SQL tasks, as they help ensure that the generated sentences are not only syntactically correct but also semantically meaningful.

A Further Discussion of Algorithm 1

As discussed in Section 3.3, we need to ensure that the generated sentences meet the criteria of containing required information and being reasonable. To ensure that the generated sentences are both compositional and complex, we use logic checks to limit the complexity of the generated examples and prevent the generation of contradictory examples. Logic checks filter out examples that are inconsistent with the input data or that do not satisfy the required conditions.

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 subquery; 2) more than one \texttt{HAVING} condition; 3) more than three \texttt{WHERE} conditions; 4) more than one \texttt{ORDER BY} clause; 5) new conditions for a subquery. On the NL side, since the substitution function in Algorithm 1 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.

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 2.2, 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 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 6 discuss some examples for ease of understanding.

B Unseen SQL Structure Template in Spider-CG

Although we limit the complexity of the generated examples lower than the upper bound of the Spider dataset, Spider-CG still contains unseen SQL structure templates. For example, the NatSQL template ‘SELECT COL WHERE COL > VAL or count(TABLE.*) >= VAL GROUP BY COL’ and corresponding SQL can not be found in the original Spider. The new templates may degrade the performance of models.

C Spider-SS Annotation Steps

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 Figure 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.

D Execution Match

The execution match metric measures whether the query results from the predicted query are the same as the gold query results. The original RATSQL can not generate the executable SQL until extending the NatSQL. The NatSQL2SQL conversion would analyze the utterance and generate executable SQL, irrelevant to the RATSQL model. Thus we only report the results of models with NatSQL. Since the execution match is similar to the exact match, we only report the top models in Table 7. Similar to the exact match, RATSQL G(S) outperform other models in most evaluation set except on the CG-APP T.
**Table 6:** Some examples of successful or unsuccessful passing the coherence checks.

<table>
<thead>
<tr>
<th>Approach</th>
<th>SpiderCG</th>
<th>CG-SUB CG</th>
<th>CG-SUB CG</th>
<th>CG-APP CG</th>
<th>CG-APP CG</th>
</tr>
</thead>
<tbody>
<tr>
<td>RATSQLEn</td>
<td>75.8%</td>
<td>86.7%</td>
<td>70.0%</td>
<td>70.4%</td>
<td>68.9%</td>
</tr>
<tr>
<td>RATSQLEn</td>
<td>74.7%</td>
<td>87.9%</td>
<td>76.4%</td>
<td>82.0%</td>
<td>72.5%</td>
</tr>
<tr>
<td>RATSQLEn</td>
<td>76.7%</td>
<td>88.3%</td>
<td>80.4%</td>
<td>78.8%</td>
<td>75.1%</td>
</tr>
</tbody>
</table>

**Table 7:** Execution match accuracy on evaluation sets.