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

Anonymous ACL submission

Abstract

Recently, the challenge of compositional generalization in NLP has attracted more and more attention. Specifically, many prior works show that neural networks struggle with compositional generalization where training and test distributions differ. However, most of these works are based on word-level synthetic data or a specific data split method to generate compositional biases. In this work, we propose a clause-level compositional example generation method, and we focus on text-to-SQL tasks. We first split the sentences in the Spider text-to-SQL dataset into several sub-sentences, then we annotate each sub-sentence with its corresponding SQL clause, resulting in our new dataset Spider-SS. Building upon Spider-SS, we further construct a new dataset named Spider-CG, by composing Spider-SS sub-sentences to test the ability of models to generalize compositionally. Experiments show that previous 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 the RATSQL+GAP model to fit 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, the 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 observation might hold even when the new samples are composed of known constituents; e.g., on SCAN dataset (Lake and Baroni, 2018), many models provide the wrong 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 data split and word substitution (Finegan-Dollak et al., 2018; Shaw et al., 2021). However, data split methods are limited by the dataset content, making it difficult to construct a challenging benchmark while ensuring that every compound appears in the training set. Ensuring a reasonable data split may also lead to a reduction in the size of the dataset, e.g., the training set drop from 7000 to 3282 in the Spider TCMD split (Yu et al., 2018b; Shaw et al., 2021).

Word substitution is often used to generate compositional variations for data augmentation (Yu et al., 2021; Andreas, 2020). However, this method cannot generate more complex sentences, i.e., cannot generate “jump twice and walk” from the data only containing “jump twice”, “walk”, and “walk twice”. In addition, in the cross-domain text-to-SQL setting, examples generated by word substitution are supposed to be considered as in-distribution data, since cross-domain models are expected to generalize to unseen domains with unseen utterances, including examples generated by word substitution.

In this work, we first introduce our Spider-SS dataset (SS stands for sub-sentence) derived from the Spider benchmark (Yu et al., 2018b). Figure 1 presents a comparison between Spider and Spider-SS. To build Spider-SS, we 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. In order to reduce the difficulty of annotation, we annotate the query in an intermediate representation called NatSQL (Gan et al., 2021b), because it is simpler and syntactically aligns better with natural language (NL).

1We will release code and dataset upon publication.
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 construct the Spider-CG benchmark for measuring the compositional generalization performance of text-to-SQL models.

- We show that the RATSQAL+GAP+NatSQL model can be adapted to sub-sentence-based text-to-SQL data, and that this improves its generalization performance.

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 because some examples are difficult to annotate using SQL. 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 a SQL query.

The purpose of building Spider-SS is to attain clause-level text-to-SQL data to generate more
complex examples through the combination of clauses. Besides, it is easier to build models with better performance based on more finely labeled data, especially in areas with relatively few resources, i.e., there are only 7000 training samples in the Spider dataset. 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
### 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 second 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 21,168 examples generated from the Spider-SS training set, while 2,966 examples are generated from the development set. In CG-APP, examples generated from training and development sets are 19,241 and 3,290, respectively. Therefore, the whole Spider-CG contains 46,665 examples, which is about six times the Spider-SS.

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 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.

#### 3.2 Generation Algorithm

Before generating the Spider-CG, we need to generate the compositional element. Considering that there are some unsatisfactory splitting results in spider-SS, in order to ensure the quality of generated sentences, compositional elements abandoned the examples with "NO MENTIONED" and "extra" column, such as the first and second examples in Figure 4. Each element contains one or more sub-sentences, but these sub-sentences must be annotated as WHERE or ORDER BY clauses. We collect the sub-sentences for compositional elements by scanning all sub-sentence from top to bottom or from bottom to top 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, we do not generate any element from the example in Figure 1.

According to Algorithm 1, we can generate the CG-SUB and CG-APP based on compositional elements. 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 1 how many times.

---

**Figure 5:** It is difficult to annotate if using the SQL instead of NatSQL.
3.3 Quality Assurance

We randomly sampled 2000 examples from the Spider-CG dataset, around 99% of which are acceptable. Acceptable does not mean that there are no grammatical errors; it means that the sentence has been clearly expressed and does not cause ambiguity. Besides, acceptable examples may be meaningless. We give two acceptable but not perfect examples in Table 1, where the first sentence contains the grammatical error. At the same time, the other is meaningless because the content it wants to query is the condition it gave. We define an acceptable example as an example where you may not find a problem without studying it carefully.

The ‘can_be_substituted_by’ and ‘can_append’ function in Algorithm 1 prevent mistakes while generating the Spider-CG. These two methods check whether the sub-sentences can be appropriately connected, e.g., whether the previous word is the same and whether the part of speech of the first word in the sub-sentence is the same. Besides, these two methods also prevent generating overly complex or contradictory SQL, e.g., ‘give me the name of the youngest student who is the oldest’.

| Ques | List the name and age of the heads of departments are older than 56? |
| SQL  | SELECT name, age FROM head WHERE age > 56 |
| Ques | Show the name of employees named Mark Young? |
| SQL  | SELECT name FROM employee WHERE name = 'Mark Young' |

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

![Figure 6: A example of encoding the whole sentence but decoding only the sub-sentence.](image)

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 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 only decoding the sub-sentence of ‘who is older than ten’ 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 he ask 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 only decoding the sub-sentence of ‘who is older than ten’ and outputting the corresponding NatSQL clauses. Based on this modification, a seq2seq text-to-SQL model can be adapted to the Spider-SS.

We modify the RATSQL+GAP+NatSQL (Wang...
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:

- **Spider**: the original Spider development set with 1,034 examples for cross-domain in-distribution text-to-SQL evaluation.
- **CG-SUB**: the CG-SUB training set, containing 21,168 examples generated from Spider-SS training set by substituting sub-sentence method. CG-SUB is used for in-domain in-distribution text-to-SQL evaluation.
- **CG-SUBD**: the CG-SUB development set containing 2,966 examples for cross-domain in-distribution text-to-SQL evaluation.
- **CG-APP**: the CG-APP training set, containing 19,241 examples generated from Spider-SS training set by appending sub-sentence method. CG-APP is used for in-domain out-of-distribution text-to-SQL evaluation.
- **CG-APPD**: the CG-APP development set containing 3,290 examples for cross-domain out-of-distribution text-to-SQL evaluation.

Our evaluation is based on the exact match and execution 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. The execution match metric measures whether the query results from the predicted query are the same as the gold query results. All models are only trained on 7000 Spider or Spider-SS training examples. We evaluate the following open-source models that reach competitive performance on Spider:

- **RATSQL**: The RATSQL+BERT model trained on Spider (Wang et al., 2020; Devlin et al., 2019).
- **CG-SUB**: The CG-SUB training set, containing 1,034 examples for cross-domain in-distribution text-to-SQL evaluation.
- **CG-SUBD**: The CG-SUB development set containing 2,966 examples for cross-domain in-distribution text-to-SQL evaluation.
- **CG-APP**: The CG-APP training set, containing 19,241 examples generated from Spider-SS training set by appending sub-sentence method. CG-APP is used for in-domain out-of-distribution text-to-SQL evaluation.
- **CG-APPD**: The CG-APP development set containing 3,290 examples for cross-domain out-of-distribution text-to-SQL evaluation.

### 5 Experiment

#### 5.1 Experimental Setup

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:

- **Spider**: the original Spider development set with 1,034 examples for cross-domain in-distribution text-to-SQL evaluation.
- **CG-SUB**: the CG-SUB training set, containing 21,168 examples generated from Spider-SS training set by substituting sub-sentence method. CG-SUB is used for in-domain in-distribution text-to-SQL evaluation.
- **CG-SUBD**: the CG-SUB development set containing 2,966 examples for cross-domain in-distribution text-to-SQL evaluation.
- **CG-APP**: the CG-APP training set, containing 19,241 examples generated from Spider-SS training set by appending sub-sentence method. CG-APP is used for in-domain out-of-distribution text-to-SQL evaluation.
- **CG-APPD**: the CG-APP development set containing 3,290 examples for cross-domain out-of-distribution text-to-SQL evaluation.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Exact Match</th>
<th>Execution Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training Set</td>
<td>89.4%</td>
<td>94.0%</td>
</tr>
<tr>
<td>Development Set</td>
<td>90.0%</td>
<td>94.3%</td>
</tr>
</tbody>
</table>

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>easy</th>
<th>medium</th>
<th>hard</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpiderD</td>
<td>24.1%</td>
<td>43.1%</td>
<td>16.8%</td>
<td>16.1%</td>
</tr>
<tr>
<td>CG-SUBT</td>
<td>28.3%</td>
<td>38.4%</td>
<td>20.8%</td>
<td>12.5%</td>
</tr>
<tr>
<td>CG-SUBD</td>
<td>33.8%</td>
<td>37.4%</td>
<td>13.6%</td>
<td>12.6%</td>
</tr>
<tr>
<td>CG-APPT</td>
<td>3.2%</td>
<td>30.3%</td>
<td>27.3%</td>
<td>39.1%</td>
</tr>
<tr>
<td>CG-APPD</td>
<td>2.3%</td>
<td>41.9%</td>
<td>22.9%</td>
<td>32.8%</td>
</tr>
</tbody>
</table>

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

- **RATSQL**: The RATSQL+GAP model trained on Spider (Shi et al., 2021).
- **RATSQL**: The RATSQL+GAP+NatSQL model trained on NatSQL (Gan et al., 2021b).
- **RATSQL**: The modified RATSQL+GAP+NatSQL model trained on Spider-SS, as discussed in Section 4.

#### 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. Most equivalent and corrected SQL cannot get positive results in exact match metric, while a small part can not either in 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 model 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\textsub 1 and CG-SUB\textsub 2 is similar to that of Spider\textsub D. The similar distributions among CG-SUB\textsub T, CG-SUB\textsub D, and Spider\textsub 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\textsub T and CG-APP\textsub D are obviously different from that of Spider\textsub 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 words 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, showing that the algorithm performs consistently for unseen datasets. 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.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Deviation &lt;= 1</th>
<th>Deviation &lt;= 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG-SUB\textsub T</td>
<td>93.2%</td>
<td>94.4%</td>
</tr>
<tr>
<td>CG-SUB\textsub D</td>
<td>92.9%</td>
<td>94.1%</td>
</tr>
<tr>
<td>CG-APP\textsub T</td>
<td>86.0%</td>
<td>90.4%</td>
</tr>
<tr>
<td>CG-APP\textsub D</td>
<td>88.9%</td>
<td>92.6%</td>
</tr>
</tbody>
</table>

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 words.

Table 5 presents the exact match accuracy on the five different evaluation sets. Specifically, the RATSQL\textsub GNS consistently outperforms other models. We use the sentence split algorithm to split every sentence before inputting the RATSQL\textsub GNS. Although there are some un-similar splitting results, it did not prevent the RATSQL\textsub GNS from getting good performance. RATSQL\textsub GNS and RATSQL\textsub GN have close results in the in-distribution dataset, i.e., Spider\textsub D, CG-SUB\textsub T, and CG-SUB\textsub D, but the RATSQL\textsub GNS significantly improves over RATSQL\textsub GN on the OOD dataset, i.e., CG-APP\textsub T and CG-APP\textsub D. The results demonstrate that the sub-sentence-based method can improve the generalization performance.

The evaluation results of all models in Spider\textsub D and CG-SUB\textsub D are close, which further confirm that the sub-sentence substitution method generates in-distribution data in a cross-domain text-to-SQL setting. In the two OOD datasets, CG-APP\textsub T and CG-APP\textsub D, the performance of all models has dropped by about 10% to 30%.

Although the performance of all models on Spider\textsub D is close, the performance of RATSQL\textsub GN and RATSQL\textsub GNS is significantly better in the rest four datasets. One of the reasons is that RATSQL and RATSQL\textsub G are trained on SQL while RATSQL\textsub GN and RATSQL\textsub GN use NatSQL for training. As discussed in Section 5.2, since the training data of Spider and Spider-SS are about 10% different, this 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, ex-
Among various model architectures (Guo 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.

### 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 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. However, this method does not support the benchmark construction for cross-domain out-of-distribution compositional generalization evaluation.

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.

### 6 Related Work

#### Text-to-SQL translation

To achieve text-to-SQL translation, researchers have built various benchmarks (Iyer et al., 2017; Ana-Maria Popescu et al., 2003; Tang and Mooney, 2000; Giordani and Moschitti, 2012; Li and Jagadish, 2014; Yaghinazadeh et al., 2017; Zhong et al., 2017; Yu et al., 2018b). In particular, most recent works focus on improving the performance on Spider benchmark (Yu et al., 2018b), where models are required to generate complex SQL in cross-domain setting. Among various model architectures (Guo et al., 2021; Zhang et al., 2019; Wang et al., 2020), most of them have implemented the pre-training method, including the latest state-of-the-art model (Scholak et al., 2021; Cao et al., 2021), where Yu et al. (2021) and Rubin and Berant (2021) augment the text-to-SQL data for pre-training.

#### 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 an 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.

### 7 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 examples, while modifying the RAT-SQL+GAP+NatSQL model to fit the segmented data of Spider-SS improves compositional generalization performance.
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.


from Natural Language using Reinforcement Learning. *CoRR*, abs/1709.0.