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

Anonymous ACL submission

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 data split (Shaw et al., 2021) and word substitution (Finegan-Dollak et al., 2018). 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 dataset size: e.g., the training set drops from 7000 to 3282 in the Spider TCMD split (Yu et al., 2018b; Shaw et al., 2021).

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 corresponding SQL clause, reducing the difficulty of

[1] We will release code and dataset upon publication.

Spider Example:

Sentence: What type of pet is the youngest animal, and how much does it weigh?

SQL: SELECT PetType , Weight FROM Pets ORDER BY Pet_Age LIMIT 1

Spider−SS Example:

SubSentence: What type of pet
NatSQL: SELECT Pets.Pettype

SubSentence: is the youngest animal
NatSQL: ORDER BY Pets.Pet_Age LIMIT 1

SubSentence: , and how much does it weigh?
NatSQL: SELECT Pets.Weight

Figure 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.

Spider−SS :

Example−1:

SubSentence: What is the name and nation of the singer
NatSQL: SELECT Singer.Name SELECT Singer.Country

SubSentence: who have a song having 'Hey' in its name?
NatSQL: WHERE Concert.Song_Name like '%Hey%'

Example−2:

SubSentence: What are the names of the singers
NatSQL: SELECT Singer.Name

SubSentence: who performed in a concert in 2014?
NatSQL: WHERE Concert.Year = 2014

Spider−CG :

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

Sentence: What is the name and nation of the singer who performed in a concert in 2014?
NatSQL: SELECT Singer.Name, Singer.Country WHERE Concert.Year = 2014

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

Sentence: What is the name and nation of the singer who have a song having 'Hey' in its name and who performed in a concert in 2014?
NatSQL: SELECT Singer.Name, Singer.Country WHERE Concert.Song_Name like '%Hey%' AND Concert.Year = 2014

Figure 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.

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 compounds. 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 24,134 examples generated by substituting sub-sentences; we consider most data in this subset as in-distribution. The second subset contains 22,531 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. Experiments with RATSQL+GAP (Shi et al., 2021) show that our Spider-CG is more challenging than the existing TMCD split (Shaw et al., 2021).

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.

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

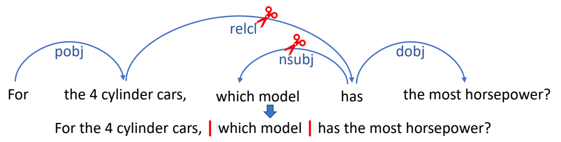

Figure 3: Dependency structure of a sentence and how to split this sentence into three sub-sentences.

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,

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

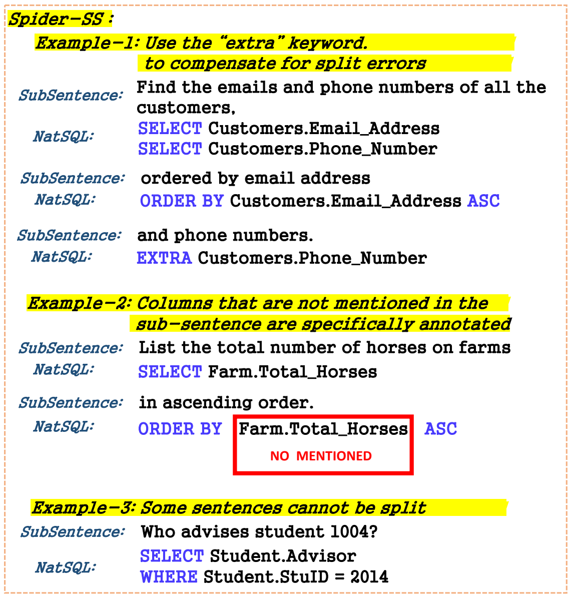

Figure 4: Spider-SS examples in three special cases.

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 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 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

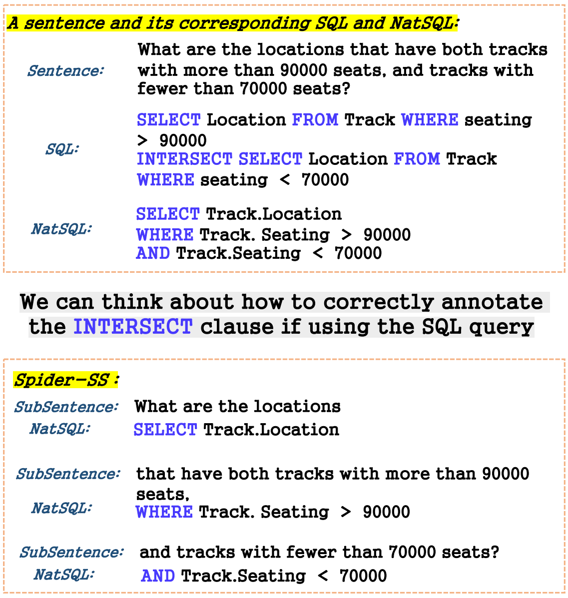

Figure 5: It is difficult to annotate if using the SQL instead of NatSQL.

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 dataset. if more data is needed, we can append sub-sentences to the CG-SUB examples.

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 domain cannot be used in another because the schema

Algorithm 1 Generate CG-SUB and CG-APP dataset in a certain domain

Input: e_list ▷ All compositional elements in a domain
Output: cg_sub and cg_app ▷ CG-SUB and CG-APP dataset in a certain domain

for Every $element_1$ in e_list **do**
    for Every $element_2$ in e_list **do**
        if $element_1$!= $element_2$ **then**
            if $element_1$.can_be_substituted_by($element_2$) **then**
                cg_sub.append($element_1$.generate_substitution_example($element_2$))
            if $element_1$.can_append($element_2$) **then**
                cg_app.append($element_1$.generate_appending_example($element_2$))
return cg_sub, cg_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.

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 SQL. 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.

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

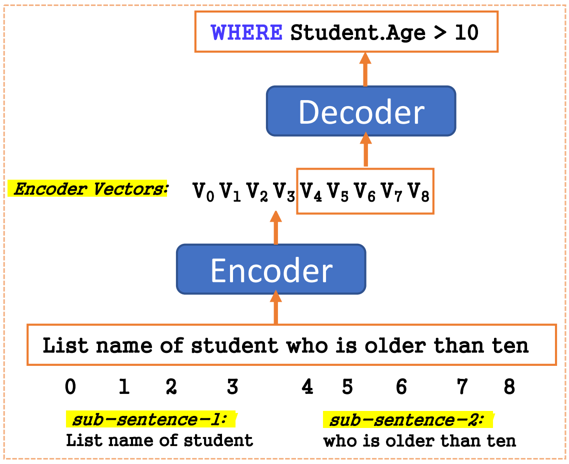

Figure 6: A example of encoding the whole sentence but decoding only the sub-sentence.

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 only decoding the sub-sentence of ‘*who is older than ten*’ and outputting the corresponding NatSQL clause. Based on this modification, a seq2seq text-to-SQL model can be adapted to the Spider-SS. Although previous span-based semantic parsers (Yin et al., 2021; Herzig and Berant, 2021) can work with aligned annotations based on the Spider-SS dataset, none of them are designed for complex text-to-SQL problems. Our modification idea is similar in principle to the span-based semantic parsers, but we did not change the existing model according to the span-based because our modification idea has a smaller workload.

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:

- **Spider$_\text{D}$**: the original Spider development set with 1,034 examples for *cross-domain in-distribution* text-to-SQL evaluation.
- **CG-SUB$_\text{T}$**: the CG-SUB training set, containing 21,168 examples generated from Spider-SS training set by substituting sub-sentences. CG-SUB$_\text{T}$ can be used for *in-domain in-distribution* text-to-SQL evaluation.
- **CG-SUB$_\text{D}$**: the CG-SUB development set containing 2,966 examples for *cross-domain in-distribution* text-to-SQL evaluation.
- **CG-APP$_\text{T}$**: the CG-APP training set, containing 19,241 examples generated from Spider-SS training set by appending sub-sentences. CG-APP$_\text{T}$ can be used for *in-domain out-of-distribution* [3] text-to-SQL evaluation.
- **CG-APP$_\text{D}$**: the CG-APP development set containing 3,290 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.
- **RATSQL$_\text{B}$**: The RATSQL model using the BERT (Devlin et al., 2019) embeddings.
- **RATSQL$_\text{G}$**: The RATSQL model using the GAP (Shi et al., 2021) embeddings.
- $_\text{(N)}$: This subscript indicates that the model use NatSQL instead of SQL.
- $_\text{(S)}$: This subscript indicates that the model is modified according to Section 4 and trained on Spider-SS. Besides, since Spider-SS is annotated by NatSQL, this subscript also indicates that the model uses NatSQL instead of SQL.

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$_\text{B(S)}$. RATSQL$_\text{B(S)}$ 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.

Dataset	Exact Match	Execution Match
Training Set	89.4%	94.0%
Development Set	90.0%	94.3%

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.

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$_\text{G}$ 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

[3] Out-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.

Dataset	easy	medium	hard	extra
Spider$_{\mathrm{D}}$	24.1%	43.1%	16.8%	16.1%
CG-SUB$_{\mathrm{T}}$	28.3%	38.4%	20.8%	12.5%
CG-SUB$_{\mathrm{D}}$	33.8%	37.4%	13.6%	12.6%
CG-APP$_{\mathrm{T}}$	3.2%	30.3%	27.3%	39.1%
CG-APP$_{\mathrm{D}}$	2.3%	41.9%	22.9%	32.8%

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

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

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.

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

Table 5: Exact match accuracy on evaluation sets.

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$_{\mathrm{T}}$ and CG-SUB$_{\mathrm{D}}$ is similar to that of Spider$_{\mathrm{D}}$. The similar distributions among CG-SUB$_{\mathrm{T}}$, CG-SUB$_{\mathrm{D}}$, and Spider$_{\mathrm{D}}$ 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$_{\mathrm{T}}$ and CG-APP$_{\mathrm{D}}$ are obviously different from that of Spider$_{\mathrm{D}}$. 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.

5.4 Model Results

Table 5 presents the exact match accuracy on the five different evaluation sets. In the two OOD datasets, CG-APP$_{\mathrm{T}}$ and CG-APP$_{\mathrm{D}}$, the performance of all models has dropped by about 10% to 30%. However, the models trained on Spider-SS significantly outperform those trained on Spider when evaluated on the OOD datasets. We use the sentence split algorithm to split every sentence before inputting the models with subscript (S). Although there are some un-similar splitting results, it did not prevent the models with subscript (S) from getting good performance, i.e., the RATSQL$_{\mathrm{G(S)}}$ consistently outperforms all other models on all evaluation sets. These results demonstrate that the sub-sentence-based method can improve the generalization performance. The limitation is that the method may not be compatible with the original model, e.g., original hyperparameters in $\text{RATSQL}_{\text{B(S)}}$ are not workable, and the performance of GNN on the Spider_{D} and D_{D} is degraded.

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

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

6 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 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. 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 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.

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.

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 subset, while modifying the models to fit the segmented data of Spider-SS improves compositional generalization performance.

References

Ben Bogin, Jonathan Berant, and Matt Gardner. 2019. Representing schema structure with graph neural networks for text-to-SQL parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4560–4565, Florence, Italy. Association for Computational Linguistics.

Xinyun Chen, Chen Liang, Adams Wei Yu, Dawn Song, and Denny Zhou. 2020. Compositional generalization via neural-symbolic stack machines. In *Advances in Neural Information Processing Systems*, volume 33, pages 1690–1701. Curran Associates, Inc.

Henry Conklin, Bailin Wang, Kenny Smith, and Ivan Titov. 2021. Meta-learning to compositionally generalize. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 3322–3335, Online. Association for Computational Linguistics.

Marie-Catherine de Marnee and Christopher D. Manning. 2016. Stanford typed dependencies manual.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. 2018. Improving text-to-SQL evaluation methodology. pages 351–360.

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

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

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

Jonathan Herzig and Jonathan Berant. 2021. Span-based semantic parsing for compositional generalization. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 908–921, Online. Association for Computational Linguistics.

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*.

Brenden Lake and Marco Baroni. 2018. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2873–2882. PMLR.

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

Qian Liu, Shengnan An, Jian-Guang Lou, Bei Chen, Zeqi Lin, Yan Gao, Bin Zhou, Nanning Zheng, and Dongmei Zhang. 2020. Compositional generalization by learning analytical expressions. In *Advances in Neural Information Processing Systems*, volume 33, pages 11416–11427. Curran Associates, Inc.

Inbar Oren, Jonathan Herzig, Nitish Gupta, Matt Gardner, and Jonathan Berant. 2020. Improving compositional generalization in semantic parsing. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 2482–2495, Online. Association for Computational Linguistics.

Ankur Parikh, Xuezhi Wang, Sebastian Gehrmann, Manaal Faruqui, Bhuwan Dhingra, Diyi Yang, and Dipanjan Das. 2020. ToTTo: A controlled table-to-text generation dataset. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1173–1186, Online. Association for Computational Linguistics.

Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar. Association for Computational Linguistics.

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

Peter Shaw, Ming-Wei Chang, Panupong Pasupat, and Kristina Toutanova. 2021. Compositional generalization and natural language variation: Can a semantic parsing approach handle both? In *Proceedings of the 59th Annual Meeting of the Association* for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pages 922–938, Online. Association for Computational Linguistics.

Peng Shi, Patrick Ng, Zhiguo Wang, Henghui Zhu, Alexander Hanbo Li, Jun Wang, Cicero Nogueira dos Santos, and Bing Xiang. 2021. Learning contextual representations for semantic parsing with generation-augmented pre-training. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(15):13806–13814.

Alane Suhr, Ming-Wei Chang, Peter Shaw, and Kenton Lee. 2020. Exploring unexplored generalization challenges for cross-database semantic parsing. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8372–8388, Online. Association for Computational Linguistics.

Bailin Wang, Mirella Lapata, and Ivan Titov. 2021. Structured reordering for modeling latent alignments in sequence transduction. In *Thirty-Fifth Conference on Neural Information Processing Systems*.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7567–7578, Online. Association for Computational Linguistics.

Hongvu Xiong and Ruixiao Sun. 2019. Transferable Natural Language Interface to Structured Queries Aided by Adversarial Generation. In *2019 IEEE 13th International Conference on Semantic Computing (ICSC)*, pages 255–262. IEEE.

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

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

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 *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1653–1663, Brussels, Belgium. Association for Computational Linguistics.

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

A Further Discussion on Algorithm 1

As discussed in Section 3.3, we need to ensure that the Spider-CG examples meet the criteria of containing required information and being reasonable. To ensure that the generated Spider-CG sentence contains the required information, the compositional element needs to contain all the information needed to derive the target NatSQL clause. Thus some sub-sentence can not be a compositional element, such as the last sub-sentence of examples 1 and 2 in Figure 4. Among them, example 1 misses *ORDER BY* information; example 2 misses *Total_Horses* column information. In contrast, the sub-sentence of the two Spider-SS examples in Figure 2 contains the required information and can be compositional elements. So, we can filter out the sub-sentences containing the "*NO MENTIONED*" and "*extra*" label, and collect the rest as compositional elements.

The '*can_be_substituted_by*' and '*can_append*' function in Algorithm 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.

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

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

Coherence checks are used to ensure that the expression of the generated sentence is coherent. As discussed in Section 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 [4] 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 Spider-SS Annotation Steps

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.

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

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

Table 6: Some examples of successful or unsuccessful passing the coherence checks.