Appendix

A Existing Synthesis Framworks

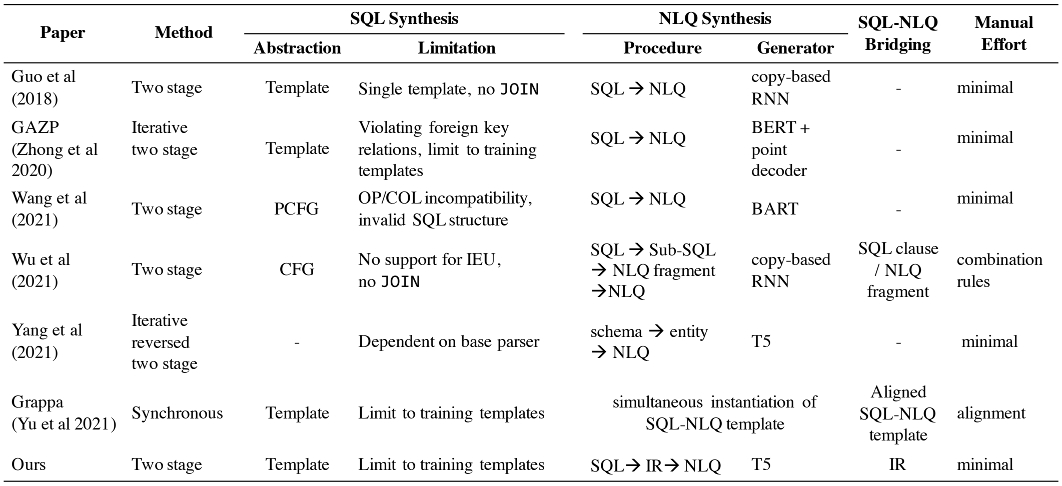
Inspired by prior work by Jia and Liang [2016] in semantic parsing, Yu et al. [2021] extended a synchronous context-free grammar (SCFG) approach to the text-to-SQL task where they manually crafted about 90 high-quality SQL-NLQ aligned patterns to generate new SQL-NLQ pairs. They found pretraining on the synthetic dataset leads to a significant improvement even tested with a very strong text-to-SQL parser RAT-SQL on the Spider benchmark.

While SCFG usually creates high-quality data because patterns are carefully designed and aligned, the coverage of the patterns is limited, and expert knowledge is required to design such patterns. Thus, more efforts are devoted to automating the procedure. Guo et al. [2018] utilized a two-stage approach by first sampling SQL queries from a simple pattern and then generating questions using a copy-based RNN encoder-decoder structure find the synthetic data that can improve the existing state-of-the-art model on the WikiSQL benchmark. Zhong et al. [2020] followed the same two-stage approach but used templates extracted from training to generate SQL and augmented the NLQ generator with pretrained transformer BERT and iteratively updated the parser and generator. Only the synthetic dataset that was created using target schemas filtered with cycle consistency can facilitate the downstream performance.

Along the same approach, Wang et al. [2021] identified problems with fixed SQL synthesis rules and employed a full-fledged probabilistic context-free grammar (PCFG) that enabled generating SQLs with varying structures. They synthesized natural language queries with a BART SQL-NLQ generator. Their synthesis method has been shown to boost the RAT-SQL parser performance on the Spider benchmark, though the improvement is not as significant as pretraining using SCFG generated synthetic data [Yu et al., 2021]. The gap might be due to the quality of the synthetic dataset as the independent selection of generation step in PCFG introduces substantial noise such as illogical SQL queries.

To improve the quality of synthetic data, Wu et al. [2021] introduced a clause-level synthesis framework: first decomposing a query into sub-clauses and translating sub-SQL clauses into sub-questions, and finally assembling sub-questions into a whole question. They found clause-based synthesis method is better than flat synthesis.

Alternatively, Yang et al. [2021] proposed to improve the quality of synthetic data by incorporating domain information in question generation. Specifically, they learned an entity sampler and synthesized questions using an entity-to-question generator with entities sampled from the sampler,



Paper	Method	SQL Synthesis		NLQ Synthesis		SQL-NLQ Bridging	Manual Effort
		Abstraction	Limitation	Procedure	Generator		
Guo et al (2018)	Two stage	Template	Single template, no `JOIN`	SQL → NLQ	copy-based RNN	-	minimal
GAZP (Zhong et al 2020)	Iterative two stage	Template	Violating foreign key relations, limit to training templates	SQL → NLQ	BERT + point decoder	-	minimal
Wang et al (2021)	Two stage	PCFG	OP/COL incompatibility, invalid SQL structure	SQL → NLQ	BART	-	minimal
Wu et al (2021)	Two stage	CFG	No support for IEU, no `JOIN`	SQL → Sub-SQL → NLQ fragment →NLQ	copy-based RNN	SQL clause / NLQ fragment	combination rules
Yang et al (2021)	Iterative reversed two stage	-	Dependent on base parser	schema → entity → NLQ	T5	-	minimal
Grappa (Yu et al 2021)	Synchronous	Template	Limit to training templates	simultaneous instantiation of SQL-NLQ template		Aligned SQL-NLQ template	alignment
Ours	Two stage	Template	Limit to training templates	SQL→ IR→ NLQ	T5	IR	minimal

Figure 3: Comparison of different data synthesis methods for text-to-SQL task. *Synchronous* refers to generating SQL and NLQ together, *Two-stage* first synthesizes SQL then generates NLQ, *reversed two-stage* first generates NLQ then synthesizes SQL. **SQL-NLQ Bridging** refers to intermediate operations or representations for matching SQL and NLQ.

followed by generating pairing SQL queries through a baseline parser. For this approach, they also attractively updated the parser and generator, in a similar fashion as in Zhong et al. [2020]. Their synthetic dataset can significantly improve a DT-Fixup parser on the Spider benchmark.

This work seeks to investigate value of synthetic dataset with current state-of-the-art PICARD model and refine a synthetic method in an automate and non-iterative manner. Thus, we examine two synthetic datasets from recent work [Wang et al., 2021, Wu et al., 2021] that demonstrate improvement of downstream performance with previous state-of-the-art text-to-SQL parser (RAT-SQL) over Spider benchmark without iterative training.

B SQL Synthesis with Schema-Weighted Column Sampling

B.1 Table Distance.

For a given database d, we first establish an undirected graph for all the tables in d. We can then compute the distance between any two tables, $e(\cdot,\cdot)$, defined as the least number of joins necessary to join the two tables under the restriction that table join can only take place with qualified primary key and foreign key information. In other words, we disable arbitrary join of two tables if they lack key and foreign key relationship.

We give some examples using one of the databases (id: `college_1`) in the Spider benchmark, as shown in Table 7.

- $e(\mathrm{T1},\mathrm{T2}) = 1$ because the column `class code` in table *class* (T1) is a foreign key in table *course* (T2). We can also observe from the table graph in Figure 4: there is a direct path between table node *class* and table node *course*.
- $e(\mathrm{T2},\mathrm{T7}) = 2$ since we first need to join table *course* (T2) with table *department* (T3), followed by joining table *department* with table *student* (T7). Note that even though we can also join using the path $T2 \rightarrow T1 \rightarrow T5 \rightarrow T7$, this is not the ***least*** number of joins between the two tables.

Table 7: Example database (id: `college_1`)

Alias	Table Name	Primary Key	Foreign Key	
			Table	Column
T1	class	`class code`	enroll	`class code`
T2	course	`course code`	class	`class code`
T3	department	`department code`	course professor student	`department code` `department code` `department code`

T4	employee	`employee number`	class
			department
			professor
				`professor employee number`
				`employee number`
				`employee number`

T5	enroll	-	-	-
T6	professor	-	-	-
T7	student	`student num`	enroll	`student number`

The reason we introduce the concept of *table distance* is that we want to leverage this value to promote table joins with appropriate relationships while discouraging illogical joins when two tables are irrelevant. During the process of column sampling, we will choose columns that have smaller table distance with the other columns that have already been selected with the objective to create more realistic synthetic SQL queries. In the example above, assume we have first sampled a column from the table *student* (T7). For the next column placeholder, we are more likely to sample a column from table *enroll* (T5) than table *professor* (T6) — it is more natural to ask questions like "how many

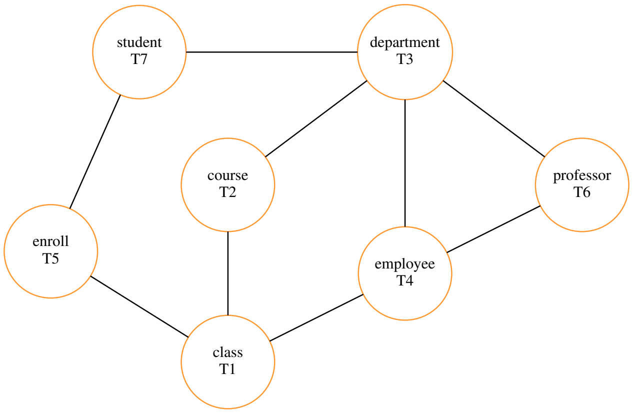


Figure 4: Example table graph (id: `college_1`)

students enrolled in class X" compared to asking "how many students enrolled in classes taught by professors who were employed before year YYYY".

B.2 Algorithm

Define a template t as $(q, \mathbf{c}, \mathbf{v})$ where q is the flat template string, $\mathbf{c} = [c_1, \ldots, c_m]$ is the set of column placeholders and $\mathbf{v} = [v_1, \ldots, v_n]$ is the set of value placeholders in q. Denote T_c to represent the table that contains column c and $S_d(\tau)$ as the set of columns in d with the *strong type* τ. Given a

Algorithm 1: Single SQL Synthesis with Schema-Weighted Column Sampling

```
Input: template $t = (q, \mathbf{c}, \mathbf{v})$, database $d$, decay rate $\gamma$
Output: SQL query $y$
Let $y = q$
Random sample $z_1$ from $S_d(\tau_{c_1})$ and replace $c_1$ with $z_1$ in $y$
Compute sampling weights
    $w(z) = \begin{cases} 1, & \text{if } T_z = T_{c_1} \\ \frac{1}{\gamma^{\delta_{c_1}(z)}}, & \text{o.w.} \end{cases}, \quad \forall z$
  where $\delta_c(z) = e(T_c, T_z)$
for $c \leftarrow c_2 : c_m$ do
    Compute sampling distribution
        $p(z) = \begin{cases} \frac{w(z)}{\sum_{z' : \tau_{z'} = \tau_c} w(z')}, & \text{if } \tau_z = \tau_c \\ 0, & \text{o.w.} \end{cases}$
    Sample $z$ from $S_d(\tau_c)$ with $p$
    Replace $c$ with $z$ in $y$
    Update sampling weights
        $w(z) \leftarrow w(z) + \begin{cases} 1, & \text{if } T_z = T_c \\ \frac{1}{\gamma^{\delta_c(z)}}, & \text{o.w.} \end{cases}, \quad \forall z$
end
for $v \leftarrow v_1 : v_n$ do
    Identify relevant columns w.r.t. $v$ and retrieve a set of possible values for $v$ from the $d$
    Random sample one value from the set and replace $v$ with the value in $y$
end
```

template t and a qualified database d, the fundamental algorithm of SQL synthesis is described in Algorithm 1 in Appendix B.

The intuition is as follows: after we select the first column for the given template, we want to choose other columns in the database that are more relevant to the first column, so as to boost the chance of synthesizing more realistic SQL queries. We do so by sampling columns, for the remaining column placeholders in the template, according to a particular sampling probability, which is a monotonically decreasing function of the table distances in the table graph for type-qualified *column candidates*, and 0 for non-qualified *column candidate*.

In Algorithm 1, the input γ is the hyperparameter that controls the decay rate in the sampling probability for qualified columns. By selecting an appropriate value for γ ($\gamma = 5$), the average table count in our synthetic data constructed from the schema-weighted column sampling method is close to that in the real Spider benchmark as shown in Figure 5, while the random column sampling mechanism tend to generate SQLs that are overly complicated. See Appendix A for the experiment details.

B.3 Value of γ in Algorithm 1.

Recall that in Algorithm 1, γ is a hyperparameter that controls the decay rate in the sampling probability for columns that are farther away from the columns that have already been selected. Under the restricted join condition, we look at the number of tables in a query as a proxy to the table distance. To determine the value of γ, we randomly sample 7000 synthetic SQL queries with replacement and calculate the average number of tables from the samples. We repeat this process for 1000 times and plot the distribution. Then we perform the same steps for the real Spider training data. We chose γ so that the distribution of the average number of tables in the synthetic data is close to the real data. This helps prevent generating over-simplified or over-complicated SQL queries.

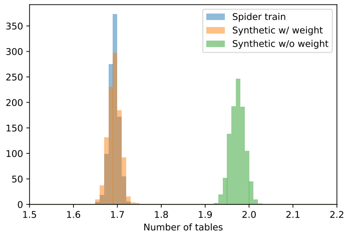


Figure 5: Histogram of the average table count (i.e. number of joins) for three types of datasets with $\gamma = 5$. Our schema-distance-weighted column sampling reduces the table number of synthetic SQLs and better matches the training distribution.

Based on this experiment, we chose γ to be 5 for the Spider benchmark. Figure 5 displays the distribution for three types of datasets: Spider training, synthetic dataset with schema-distance-weighted column sampling, and synthetic dataset with random column sampling. The figure demonstrates that the weighted sampling process, which provides an interface to tune the value of γ, can generate synthetic SQL queries that better match the real training data.