T5QL: Taming language models for SQL generation

Samuel Arcadinho, David Aparício, Hugo Veiga, António Alegria
Outsystems
{samuel.arcadinho, david.aparicio, hugo.veiga, antonio.alegria}@outsystems.com

Abstract

Automatic SQL generation has been an active research area, aiming at streamlining the access to databases by writing natural language with the given intent instead of writing SQL. Current state-of-the-art (SOTA) methods for semantic parsing depend on large language models (LLMs) to achieve high predictive accuracy on benchmark datasets. This reduces their applicability, since LLMs require expensive GPUs. Furthermore, SOTA methods are ungrounded and thus not guaranteed to always generate valid SQL. Here we propose T5QL, a new SQL generation method that improves the performance in benchmark datasets when using smaller LMs, namely T5-Base, by $\approx$ 13pp when compared against SOTA methods. Additionally, T5QL is guaranteed to always output valid SQL using a context-free grammar to constrain SQL generation. Finally, we show that dividing semantic parsing in two tasks, candidate SQLs generation and candidate re-ranking, is a promising research avenue that can reduce the need for large LMs.

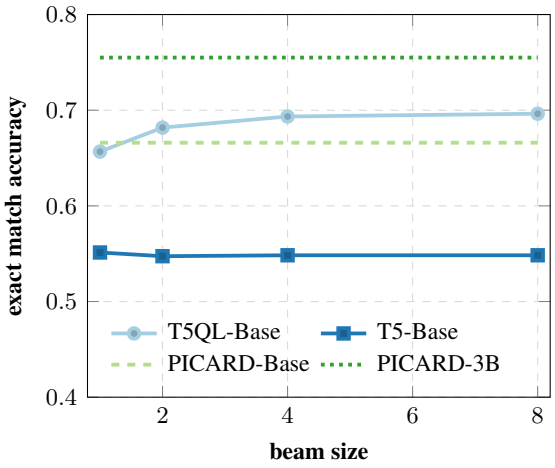

Figure 1: Exact-match accuracy of the highest scoring prediction as a function of beam size on the Spider development set. Our method, T5QL, significantly improves upon T5-Base and is superior to PICARD-Base. PICARD-3B remains the SOTA for very large LMs, i.e., PICARD-3B uses T5-3B which is $\approx 13x$ larger than T5-Base. Results for PICARD-Base and PICARD-3B are straight (dashed) lines since Scholak et al. (2021) only report results in the setting using database content for a single point, namely beam search with 4 beams.

1 Introduction

Automated code generation has long been considered one of the fundamental tasks in computer science (Pnueli and Rosner, 1989). Recently, deep learning (DL) methods for code generation have been proposed which overcome the lack of flexibility of more traditional approaches (Le et al., 2020). Some DL approaches can act as code completion tools (Svyatkovskiy et al., 2020; Chen et al., 2021) while others can use natural language (NL) as input to generate code (Yin and Neubig, 2017), i.e., semantic parsing (Kamath and Das, 2018). The latter methods are particularly helpful for developers that are not proficient in all programming languages that are part of their development pipeline. For example, a developer might be familiar with the controller language (e.g., Python) but unfamiliar with the database access language (e.g., SQL).

Generating SQL from NL is challenging because the NL query might be ambiguous (e.g., columns from different tables can have the same name). Futhermore, obtaining labelled pairs of NL queries to SQL is hard, time-consuming, and requires labellers that are proficient in SQL. In recent years, benchmark datasets have been used by developers to evaluate their methods, namely Spider (Raffel et al., 2019) and CoSQL (Yu et al., 2019).

PICARD (Scholak et al., 2021) is the current SOTA method, i.e., the highest ranked method on Spider. It is built on top of T5 (Raffel et al., 2019), a general purpose LLM. As proven by Merrill et al. (2021), LLMs are ungrounded and thus can generate any token at any given step, which may result in invalid SQL; thus, to improve upon T5, PICARD

Proceedings of the 2nd Workshop on Natural Language Generation, Evaluation, and Metrics (GEM), pages 276 - 286

prunes the search tree in order to avoid generating invalid SQL. However, since PICARD fully prunes branches during beam search, it is not guaranteed to always generate an answer. Another major issue with PICARD is that it needs a very large LM to achieve good performance: PICARD gets $\approx$ 75.5% exact match (EM) accuracy in Spider's development set when using T5-3B, but only $\approx$ 66.6% when using the smaller T5-Base.

Here, we propose T5QL, a novel SQL generation method that achieves 69.3% EM on Spider development set using T5-Base instead of the $\approx$13x larger T5-3B. T5QL uses constrained decoding to ensure that it always generates valid SQL, and it always generates an answer. Our main contributions are:

1. Narrow the gap between large and small LMs (Figure 1). With beam size equal to 4 and using T5-Base, T5QL achieves 69.3% EM accuracy on Spider, versus 66.6% obtained by PICARD. PICARD with T5-3B is still SOTA (75.5%) but it requires much larger GPUs, which are expensive and thus not available for regular practitioners.

2. Propose a constrained decoding method that always generates valid SQL, except for infrequent model hallucinations. In Appendix A.1 we show one such case.

3. Propose a novel ranker model for SQL generation. This model re-ranks the generator model's predictions after beam search, boosting EM on Spider for larger beam sizes (e.g., 8 beams) from 67.9% to 69.6%.

The remainder of the paper is organized as follows. Section 2 presents SOTA for SQL generation. Section 3 describes T5QL's main components, namely constrained decoding and the ranker. Section 4 shows our results. Finally, Section 5 concludes our work.

2 State-of-the-art

Automated program generation has long been one of the major goals of computer science. Various program synthesis tools have been proposed that generate SQL from code fragments (Cheung et al., 2012) or pairs of input-output examples (Orvalho et al., 2020). However, code fragments might not be readily available if the developer does not write code or does not want to, and creating enough input-output examples for the program synthesis tool to be effective might be cumbersome. Other tools generate SQL from NL which is more developer-friendly (Yaghmazadeh et al., 2017).

The complexity of generating SQL from NL varies with the length and complexity of the SQL query and the size of the database schema. Thus, in order to properly evaluate and compare methods' performance, multiple benchmark datasets have been proposed, namely Spider (Raffel et al., 2019), Spider-SSP (Shaw et al., 2021), and CoSQL (Yu et al., 2019). We describe these benchmarks in detail in Section 4.2 and discuss how they relate to our research questions (enumerated at the start of Section 4).

The current SOTA for SQL generation (i.e., the methods that achieve the highest performance on benchmark datasets) comprises DL methods. DL methods for code generation avoid the complexity of traditional program synthesis and, thus, are generally faster during generation (Parisotto et al., 2016; Hayati et al., 2018; Sun et al., 2019).

RatSQL's authors argue that predicting SQL directly from NL is hard and can be made easier by instead predicting an intermediate representation (IR) that is more similar to NL than SQL is (Wang et al., 2019; Gan et al., 2021). With this insight, they obtained SOTA results on Spider. However, their IR is not capable of representing all SQLs and, thus, for some queries the correct SQL is not obtainable, leading to a loss of EM accuracy. Other approaches were built on top of RatSQL with good results (Zhao et al., 2021; Shi et al., 2020; Yu et al., 2020). One of the major disadvantages of these methods is that, since they use custom architectures, they cannot leverage pre-trained LLMs in their decoding step. Being able to leverage LLMs is beneficial since they can be used for multiple tasks. For example, Xie et al. (2022) unifies structured knowledge grounding tasks into a text-to-text format and are thus able to train the same model for different tasks.

To the best of our knowledge, Shaw et al. (2021) were the first to propose a method that uses an LLM, namely T5, and evaluate it on Spider. They concluded that their method had good predictive capabilities, but sometimes generated syntactically incorrect SQL and had lower precision in out-of-distribution examples. Since T5 is ungrounded, it cannot be guaranteed to always generate valid SQL; the same is true for other LLMs (Merrill et al., 2021). In order to address the issue, Xiao et al.

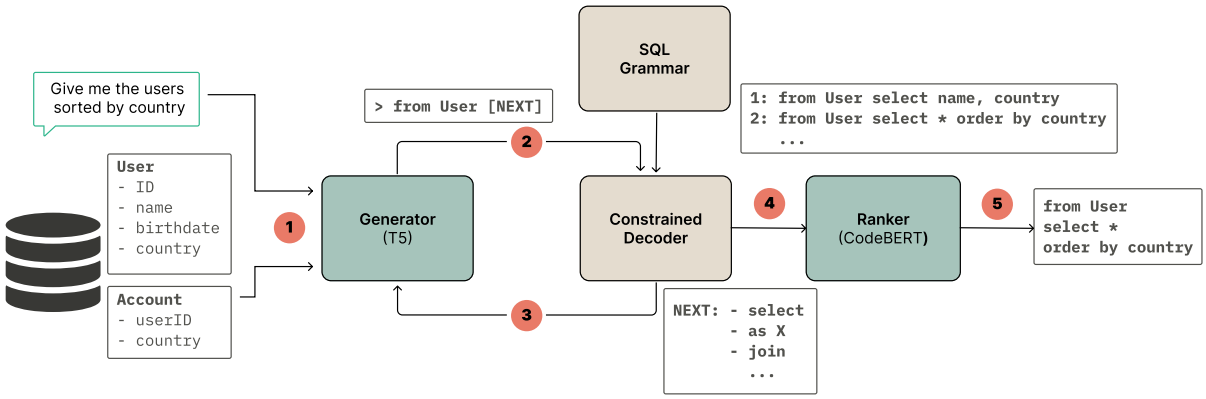

Figure 2: T5QL model architecture. T5QL receives as input an NL query and a database schema (step 1). Then, the generator model, T5, consults the constrained decoder to know which tokens are valid (step 2) and predicts the next token (step 3). This step is done iteratively. The generation is done using beam search, thus producing a set of k candidates which are given as input to the ranker model (step 4). Finally, the ranker model ranks all candidates and a final prediction is outputted by T5QL (step 5).

(2016) propose a method that constrains the output generation based on grammatical rules. They also compare a model trained with constraints and verify that using the constraints only during inference improves the model.

Targeting code generation specifically, Scholak et al. (2021) propose PICARD, a method that constrains the model generation by removing wrong outputs during beam search. By doing so, PICARD is the current SOTA in the Spider benchmark. However, they report that PICARD did not generate any SQL for 2% of the queries. Poesia et al. (2022) improve LLM performance in the few-shot setting by introducing two components, one that selects the examples to be given to the model and another that constrains the generation of syntactically correct SQL. However, fine-tuned models (e.g., PICARD) still perform better in the general task than their model, which was trained in the few-shot setting.

In this work, we use one model to generate candidates, a *generator*, and another to re-rank them, a *ranker*. This choice is motivated by recent work (Chen et al., 2021) where the authors show that a re-ranking method boosted performance for code generation. Regarding semantic parsing more concretely, Ye et al. (2022) use a ranker model to select candidates, and then a fine-tuned model generates the final output; their model shows good generalization capabilities and outperforms previous methods for question answering on knowledge graphs. More recently, Krishna et al. (2022) argue that when current LLMs are given a prefix prompt they can often generate text that is incoherent with the prefix. They propose a ranker model that scores the generator's candidates for an input prefix and obtain results that outperform earlier models in both automatic and human evaluation.

3 Method

We start this section by presenting an overview of our method and its architecture (Figure 2). Then, we focus on each of its main components, namely constrained decoding and the ranker. Finally, we discuss the scoring function and evaluation metrics.

3.1 Overview

Our method outputs the corresponding SQL query for a given NL query and a database schema. The database schema comprises a list of tables and their respective columns. Figure 2 shows a simple NL query, "Give me the users sorted by country", and a toy database schema with only two tables, `User` and `Account`. The generator, T5, receives the NL and the database schema as input and, starting with an empty string, it iteratively predicts the next token. However, unlike regular T5, the next token prediction is limited by the constrained decoder to only consider tokens that form a valid SQL query up to that point. For example, if the current query is `"from User"`, the next valid tokens include `"select"`, `"as X"`, and `"join"`, but do not include `"from"` or `"User"`. We discuss why we invert the *from* and the *select* statements in Section 3.2. We use beam search to generate multiple candidate queries, which are given as input to the ranker model.

3.2 Constrained decoding

We use constrained decoding to limit which tokens are considered by the generator to make the next token prediction. In order to enforce valid tokens, we build a context-free grammar (CFG) of SQL statements. Our constrained decoding method, described in Algorithm 1, is similar to the one proposed by Poesia et al. (2022): for each decoding step, given the current generation P, T5QL finds the maximum parsable prefix P^*, this means that all SQL tokens in the prefix P^* have valid syntax (lines 2–5). Then, using the lookahead feature of the parser, T5QL tokenizes all possible suffixes for P^* and adds them to trie T (lines 6–10). Finally, T5QL computes possible generation tokens by searching the possibles suffixes for P in T (lines 11–12).

Algorithm 1 Constrained decoding

1: **procedure** NEXTTOKEN(P, T) ▷ P is the current SQL generation and T the current trie
2: $P^* \leftarrow$ FINDPARSABLEPREFIX(P)
3: $S \leftarrow$ GETPARSERSTATE(P^*)
4: $N \leftarrow$ PARSERNEXTTOKENS(S)
5: $N^* \leftarrow$ FILTERWRONGTOKENS(S, N)
6: **for** n in N^* **do**
7: $C \leftarrow P^* + n$
8: $C^T \leftarrow$ SENTENCETOKENIZER(C)
9: $T \leftarrow$ ADDTOTRIE(T, C^T)
10: **end for**
11: $P^T \leftarrow$ SENTENCETOKENIZER(P)
12: **return**GETCHILDREN(T, P^T)
13: **end procedure**

We note that, while our grammar is context free, our constrained decoding method uses context to make decisions: FILTERWRONGTOKENS (line 5 of Algorithm 1) constrains the SQL generation by only allowing the generation of columns that are defined in the *from* statement and by mapping table aliases to the original tables. We should point out that, while this is currently not performed by our method, we could extend constrained decoding to enforce more rules, such as only allowing tables to be joined using valid foreign keys or limiting the *where* statement to only have conditions that have the proper return type given the column types (e.g., if a column `"X"` is of type *string*, `"X > 10"` is not a valid generation).

Next, we focus on the grammar. For brevity, we only show higher-level statements below; the entire grammar is shown in our Codalab page[1]. Statements inside square brackets indicate that they are optional (e.g., a SQL query can have an empty *where* statement).

⟨sql⟩ ⊨ ⟨expr⟩
⟨expr⟩ ⊨ ⟨query⟩ |
⟨expr⟩ `union` ⟨expr⟩ |
⟨expr⟩ `intersect` ⟨expr⟩ |
⟨expr⟩ `except` ⟨expr⟩
⟨query⟩ ⊨ `from` ⟨from-expr⟩
`select` ⟨select-expr⟩
[`where` ⟨where-expr⟩]
[`group by` ⟨groupby-expr⟩]
[`having` ⟨having-expr⟩]
[`order by` ⟨orderby-expr⟩]
[`limit` ⟨limit-expr⟩]

Our grammar only supports SQL *select* statements since our focus are queries that retrieve data from a database. These *select* statements can be a single query or contain subqueries joined by *unions*, *intersects*, and *excepts*. We note that the *from* and the *select* statements are inverted. This is done because, besides restricting T5 to only generate syntactically correct SQL, we also restrict it to only generate SQL with valid table names (i.e., tables that exist in the database schema) and valid column names (i.e., columns that exist in the database schema for the given table). To restrict the generation to only valid columns, it is helpful to first know the valid tables, which are obtained in the *from* statement. Thus, T5QL first parses the *from* statement and stores the selected tables; then, when the *select* statement is parsed, T5QL already knows what columns are valid since they had to appear in the selected tables (e.g., from the example from Figure 2, if the current query is `"from User select"`, then `"user.ID"` and `"user.name"` are valid token predictions while `"account.country"` and `"account.userId"` are not).

For a given query and database schema pair, we augment the grammar shown previously with two extra rules specifying the valid tables and the valid columns. For the example from Figure 2, we would add the following production rules:

[1] https://worksheets.codalab.org/worksheets/0x0049b642db90440e9eaf9cf6a850b4c9

```
⟨table-name⟩    ⊨  user | account
⟨column-name⟩   ⊨  user.id | user.name |
                   user.birthdate |
                   user.name |
                   user.country |
                   account.userId |
                   account.country
```

When a table has an an alias, we add one expression for the alias and another for the original table table (e.g., for a column `"alias1.columnA"`, we add two expressions to the ⟨column-name⟩ rule: `"alias1.columnA"` and `"tableX.columnA"`, assuming that `alias1` corresponds to `tableX`).

The grammar is given as input to the Lark parser[2]. We use Lark since it is one of the fastest parsers for Python, and it includes a look-ahead feature that we require.

3.3 Ranker

We use beam search to generate a set of k candidate queries and employ a ranker model to choose the best option among the k candidates. We hypothesize that splitting the task of SQL generation into two tasks, (1) SQL candidates generation and (2) SQL candidate ranking, boosts the performance of the complete task since each model is only focused on a simpler task.

We use a trained generator model to generate the dataset to train the ranker model. The T5 model described in Section 3.2 samples 16 SQL queries for each input (NL query and database schema pair) in the training dataset using beam search. From the 16 generated SQLs we sample the 12 with lowest tree edit distance (TED) (discussed in Section 3.5) to guarantee that we select hard negative examples. If the generator model does not predict the correct SQL in any of the 12 SQLs samples, we discard the one with the highest TED and add the correct SQL as one of the samples. Using the same sampling strategy (i.e., based on TED), we sample an additional two SQLs from the training dataset pertaining to the same database as the input, for a total of 14 SQLs for each input.

For the ranker model, we fine-tune CodeBERT (Feng et al., 2020) in a cross encoder setting. The ranker is given pairs of NL and SQL and predicts the probability of the pairs being correct, i.e., the SQL corresponding to the NL. We also append the terminals found in the NL using the method proposed by Lin et al. (2020) to the final NL (e.g., for the NL "People from 'France'", the NL is transformed into "People from 'France' | France").

3.4 Scoring Function

Similarly to Yee et al. (2019), we compute the final prediction score for a given input by combining the generator's probability score with the ranker's probability score using the linear combination shown in Equation 1, where t is the length of the SQL, and λ is a tunable weight. In order to compare the generator's probability $p(y|x)$ with the ranker's probability $p(x, y)$, we scale the generator's probability by t.

$$\frac{1}{t} \log p(y|x) + \lambda \log p(x, y) \quad (1)$$

3.5 Evaluation metrics

The most commonly used evaluation metrics for SQL comparison are EM and execution match (EX). EM checks if two SQLs are syntactically equivalent, while EX checks if running two SQLs yields the same output. While desirable, EX is more computationally expensive than EM since it requires running the SQL statements, which might not even be possible if we do not have access to the database content. When measuring the method's performance, it is also relevant to highlight if it also predicts terminal values or not; T5QL generates the full SQL query, including terminal values.

Since EM is binary, its value might not be very informative for the user nor the model. Partial matches sub-divide the comparison to only portions of the SQL statement, such as the *from* clause or the *where* clause. Thus, one SQL prediction might be wrong in multiple parts of the query, and this more granular information can be useful to improve the model. However, these measures are still coarse; thus, we use the TED in some experiments (namely in the ranker) when we want more information on the difference (or distance) between two SQLs.

In order to compute the TED between two SQL statements, we transform each SQL statement into a tree and use APTED[3] to compute the TED between two trees. Due to SQL's semantics, we first normalize the SQL to a canonical representation

[2] https://github.com/lark-parser/lark

[3] https://github.com/DatabaseGroup/apted

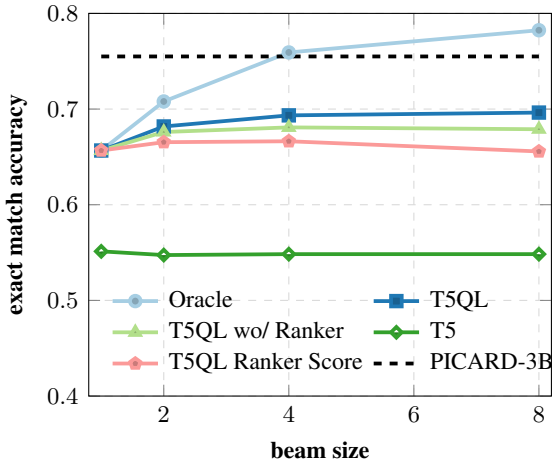

Figure 3: EM accuracy in Spider's development set by beam size. All methods use T5-Base as their LM except for PICARD-3B which uses T5-3B. The performance of PICARD-3B is shown as a straight line since the authors only report results on the development set using database content for a single point (beam search with 4 beams). The Oracle plot shows the performance ceiling for T5QL, i.e., the performance of T5QL using a perfect ranker that always outputs the correct SQL if the generator offers it as one of the candidates after beam search.

(e.g., sort the list of tables in the *select* alphabetically, transform *left joins* into *right joins*). Then modify APTED to guarantee that the TED is meaningful (e.g., the cost of removing a terminal and column name should be the same).

4 Experiments

We start by describing the experimental setup in Section 4.1. Then, we detail each dataset and the relevant evaluation metrics in Section 4.2. Then, each subsequent section (Section 4.3–4.6) tries to answer each of the following research questions:

Q1. Does constrained decoding improve the generator's performance?

Q2. Does T5QL have compositional generalization capabilities?

Q3. Does T5QL generalize to the conversational setting?

Q4. Instead of using a very large generator, can we improve performance using a ranker?

4.1 Experimental setup

For our experiments we use a G4DN Extra Large AWS machine, which has an NVIDIA T4 Tensor Core GPU installed and 4 CPU-cores. We make our code available in our public Codalab page[4][5].

4.2 Datasets

We evaluate T5QL on three benchmark datasets: Spider (Raffel et al., 2019), Spider-SSP (Shaw et al., 2021), and CoSQL (Yu et al., 2019).

Spider comprises 10,181 NL and database schemas pairs, on 200 different database schemas. Evaluation on Spider consists of two main leaderboards: EX with terminal values and EM without terminal values. At the time of writing, PICARD is the current SOTA method on both leaderboards.

Spider-SSP is a different splitting of the Spider dataset, with the aim of testing compositional generation instead of cross-database generalization, i.e., in the original Spider data split, a database schema seen in train is not seen in eval or test. Splits in the Spider-SSP dataset are made in three different fashions: random split, a split based on source length, and a split based on Target Maximum Compound Divergence (TMCD). The goal here is to evaluate if the model can have good performance on queries that it has not seen in training.

While Spider consists of a single NL and domain model pair mapped into a single SQL query, CoSQL consists of a conversational dataset with multiple iterations of NL plus data model being mapped to a SQL query. The goal of CoSQL is to simulate a user progressively exploring a data model. CoSQL contains 4,298 interactions and $\approx 12,000$ questions, on the same 200 data models used in Spider. Evaluation is done using EM without terminal values and reported using two different metrics: question match (QM) and interaction match (IM). QM evaluates if all SQLs are correctly predicted, while IM evaluates if the questions for the same interaction are correctly predicted.

4.3 Q1. Constrained decoding

LMs are unconstrained and thus can generate any token at any given time. For SQL generation, LMs may generate SQL that are syntactically incorrect, which impact their performance.

Here, we compare the performance of an unconstrained LM against T5QL without the ranker component. Both methods use the same LM, namely T5-Base, and are trained using the same training

[4] https://worksheets.codalab.org/worksheets/0x0049b642db90440e9eaf9cf6a850b4c9

[5] We will make the code available in github after the blind review process is finalized.

configuration; the only difference is that T5QL uses constrained decoding as described in Section 3.2. Both methods serialize the database schema as a string and append it to the source sequence similarly to Suhr et al. (2020). Similarly to Scholak et al. (2021), we train both methods for a maximum of 512 training epochs with mini batch size of 5, 205 gradient accumulation steps, with a learning rate of $1e^{-4}$, and an adafactor optimizer with epsilon set as $1e^{-6}$. We evaluate the models using beam search with 1, 2, 4, and 8 beams. Contrary to Scholak et al. (2021), we report results for a batch size of 1025 instead of 2048 since it lead to better results in our case.

Figure 3 shows the performance of several methods and those results are discussed in this subsection and in the next ones. All methods use T5-Base as its LLM, except for PICARD which is the current SOTA and uses T5-3B, a much larger LM.

From Figure 3, we observe that T5 achieves $\approx$ 55.1% EM accuracy using one beam, and its performance does not improve with the beam size. Our method, T5QL, without the ranker component (i.e., T5QL wo/ Ranker in Figure 3) achieves $\approx$ 65.7% EM accuracy using one beam, a gain of $\approx$ 10.6pp, which is a relative gain of $\approx$ 19.2%. Using 2 and 4 beams, we improve T5QL's performance to $\approx$ 67.6% and $\approx$ 68%, a gain of $\approx$ 1.9pp and $\approx$ 2.3pp, respectively, when compared against T5QL using only one beam. We observe a loss of performance when using 8 beams. These results highlight the advantage of using constrained decoding for SQL generation: by using a CFG to guarantee that the LM always generates valid SQL, we improve the model's performance.

4.4 Q2. Compositional generalization

Compositional generalization of LLMs has attracted attention in recent years. Shaw et al. (2021) propose Spider-SSP, a dataset that can be used to measure the compositional generalization of SQL generation methods. In this section we use Spider-SSP to evaluate if constraint decoding increases the compositional generalization capabilities of T5QL.

Shaw et al. (2021) already reported that T5-Base model struggles in most splitting strategies, particularly when using length-based split and TMCD split; we reproduce those results in Table 1 in rows T5-Base and T5-3B. We note that, in their experiments, the predicted SQL follows the convention of predicting first the *select* statement and then the *from* statement. As discussed in Section 3.2, T5QL first predicts the *from* statement and then the *select* statement. Thus, we evaluate two different models: T5-Base, which is similar to the model evaluated by Shaw et al. (2021), and T5QL-Base wo/ CD which is T5QL without the constrained decoding component (and without the ranker). We compare these models against T5QL-Base and T5-3B; the latter also predicts the *select* statement first.

We observe that T5QL-Base wo/ CD obtains significantly higher EM than T5-Base, namely for the TMCD split where there is a gain of 22pp, which is a 52% relative gain. These results highlight that predicting the tables before predicting the columns seems to help the model. This result corroborates the results obtained by Lin et al. (2020), which use a representation similar to ours. We also verify that T5QL-Base slightly, but consistently, improves upon the results obtained by T5QL-Base wo/ CD for all splitting strategies, namely in TMCD where there is a gain of 2pp. Finally, we conclude that our strategy narrows the performance gap between the performance of methods using small LMs (i.e., T5-Base) and very large LMs (i.e., T5-3B) by comparing the performance of T5QL-Base against T5-3B.

4.5 Q3. Generalize to conversational setting

Often users might want to explore their data without having to write SQL. Thus, a conversational setting where user's iteratively ask questions to an AI is particularly interesting. Yu et al. (2019) propose a dataset comprised of multiple question-SQL pairs, each consisting of several user interactions. They evaluate SQL generation methods using QM and IM. In this section we use CoSQL to evaluate if constrained decoding increases the performance of T5SQL in the conversational setting.

We observe gains of $\approx$ 7.9% and $\approx$ 5.5% in QM

	Spider-SSP			
Model	Rand.	Templ.	Len.	TMCD
T5-Base	76.5	45.3	42.5	42.3
T5QL-Base wo/ CD	84.7	58.3	50.6	64.4
T5QL-Base	85.7	61.1	54.4	65.9
T5-3B	85.6	64.8	56.7	69.6

Table 1: EM accuracy in the Spider-SSP dataset using different splitting strategies. T5QL-Base wo/ CD (i.e., without constrained decoding) and T5QL-Base adopt the strategy of predicting SQL with the *from* statement before the *select* statement, while T5-Base and T5-3B do the opposite, which is the default.

	CoSQL	
Model	QM	IM
T5QL-Base wo/ CD	42.8	14.8
T5QL-Base	50.7	20.3
PICARD - 3B	56.9	24.2

Table 2: QM and IM in the CoSQL development set.

and IM, respectively, when we add constrained decoding to T5QL-Base. We observe that PICARD-3B is still SOTA for the task, but the gap is significantly narrower. This is further evidence that constrained decoding can improve the performance of LMs in multiple SQL generation tasks.

4.6 Q4. Ranker

Current SOTA methods, such as PICARD, use beam search to find the best candidate and output it as the final prediction. Here we test whether we can boost predictive performance by, instead of using the beam-selected best candidate as the final prediction, having a ranker that finds the best candidate among the list of candidate predictions found by the generator.

Our first step to validate this hypothesis is to run beam search for multiple beam sizes k, namely 1, 2, 4, and 8, and measure the accuracy@k. In our setting, the accuracy@k can be regarded as an *oracle* ranker than can always find the correct candidate if it is present in the list of candidate generations. From Figure 3 we observe that this *oracle* could achieve 78.2% EM accuracy with 8 beams, surpassing the performance of PICARD-3B but using T5-Base instead of T5-3B, which is highly desirable due to T5-3B's expensive nature in terms of GPU costs. Thus, our goal here is to build a ranker model that can boost the performance of T5QL without the ranker (T5QL wo/ Ranker in Figure 3) and approximate it to the *oracle*'s performance.

We note that the ranker model should be of a comparable size to the generator, i.e., fit in the same GPU. Otherwise, the advantage of using a small LM as the generator is lost since we assume that the practitioner has hardware that can fit the larger ranker, which might not be true. Here we use T5-Base as the generator and CodeBERT as the ranker, which are of comparable size.

To train the ranker model, we first create a dataset following the steps described in Section 3.5. Then, we fine-tune a CodeBERT model for 50,000 training steps, using a batch size of 32 and 1 gradient accumulation step, with a $1e^{-3}$ learning rate and an AdamW optimizer with weight decay of $1e^{-2}$ and a linear schedule with warmup of 1% of the steps. We use Equation 1 to score the generated SQL; we conduct hyperparameter tuning for λ and conclude that $\lambda = 2e^{-2}$ performs best.

We analyze if combining the generator's score with the ranker's score is superior to using each of the score's individually. From Figure 3 we conclude that combining the ranker model's score with the generator model's score (i.e., the T5QL plot) improves the best EM from 67.9% to 69.3% when compared against T5QL without the ranker score. Furthermore, we also observe that using only the ranker score (i.e., the T5QL Ranker Score plot) leads to a drop in performance even when compared against T5QL wo/ Ranker. This effect is more noticeable for larger beam sizes, which indicates that the ranker model struggles to differentiate the correct SQL from the wrong SQL.

From these experiments, we conclude that the ranker boosts the performance of the generator. However, the ranker's score needs to be combined with the generator's score to guarantee that the ranker's score does not completely dominate the generator's predictions. We should also note that there is a very large gap between our ranker and the oracle, which leaves room for future research to improve the ranker model. We believe that this a promising line of research that can further narrow the gap between the performance between small LMs and large LMs.

Finally, we run T5SQL on Spider's test set and obtain 66.8% EX and 65.9% EM. These results rank among the top-10 best models in terms of EX, and as the 22nd best in terms of EM[6], whilst using small models. Small models have the advantage of being less computationally expensive and allowing more easily for the use of ensemble methods.

5 Conclusion

Here we put forward T5QL, a new method for SQL generation with SOTA performance on benchmark datasets when using small LMs. T5QL uses constrained decoding to improve predictive performance and also to guarantee that the generated SQL is always valid. Futhermore, we complement the generator model with a ranker model that is capable of choosing the best candidate SQL from a pool of a few candidates.

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

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2012. Inferring sql queries using program synthesis. *arXiv preprint arXiv:1208.2013*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *arXiv e-prints*, page arXiv:2002.08155.

Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R Woodward, John Drake, and Qiaofu Zhang. 2021. Natural sql: Making sql easier to infer from natural language specifications. *arXiv preprint arXiv:2109.05153*.

Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation. *arXiv preprint arXiv:1808.10025*.

Aishwarya Kamath and Rajarshi Das. 2018. A survey on semantic parsing. *arXiv preprint arXiv:1812.00978*.

Kalpesh Krishna, Yapei Chang, John Wieting, and Mohit Iyyer. 2022. Rankgen: Improving text generation with large ranking models. *arXiv preprint arXiv:2205.09726*.

Triet HM Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Computing Surveys (CSUR)*, 53(3):1–38.

Xi Victoria Lin, Richard Socher, and Caiming Xiong. 2020. Bridging textual and tabular data for cross-domain text-to-sql semantic parsing. *arXiv preprint arXiv:2012.12627*.

William Merrill, Yoav Goldberg, Roy Schwartz, and Noah A Smith. 2021. Provable limitations of acquiring meaning from ungrounded form: What will future language models understand? *Transactions of the Association for Computational Linguistics*, 9:1047–1060.

Pedro Orvalho, Miguel Terra-Neves, Miguel Ventura, Ruben Martins, and Vasco Manquinho. 2020. Squares: a sql synthesizer using query reverse engineering. *Proceedings of the VLDB Endowment*, 13(12):2853–2856.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*.

Amir Pnueli and Roni Rosner. 1989. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190.

Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227*.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. Picard: Parsing incrementally for constrained auto-regressive decoding from language models. *arXiv preprint arXiv:2109.05093*.

Peter Shaw, Ming-Wei Chang, Panupong Pasupat, and Kristina Toutanova. 2021. Compositional generalization and natural language variation: Can a semantic parsing approach handle both? *ArXiv*, abs/2010.12725.

Peng Shi, Patrick Ng, Zhiguo Wang, Henghui Zhu, Alexander Hanbo Li, Jun Wang, Cicero Nogueira dos Santos, and Bing Xiang. 2020. Learning Contextual Representations for Semantic Parsing with Generation-Augmented Pre-Training. *arXiv e-prints*, page arXiv:2012.10309.

Alane Laughlin Suhr, Kenton Lee, Ming-Wei Chang, and Pete Shaw. 2020. Exploring unexplored generalization challenges for cross-database semantic parsing.

Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A grammar-based structural cnn decoder for code generation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 7055–7062.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2019. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. *arXiv preprint arXiv:1911.04942*.

Chunyang Xiao, Marc Dymetman, and Claire Gardent. 2016. Sequence-based structured prediction for semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1341–1350, Berlin, Germany. Association for Computational Linguistics.

Tianbao Xie, Chen Henry Wu, Peng Shi, Ruiqi Zhong, Torsten Scholak, Michihiro Yasunaga, Chien-Sheng Wu, Ming Zhong, Pengcheng Yin, Sida I. Wang, Victor Zhong, Bailin Wang, Chengzu Li, Connor Boyle, Ansong Ni, Ziyu Yao, Dragomir R. Radev, Caiming Xiong, Lingpeng Kong, Rui Zhang, Noah A. Smith, Luke Zettlemoyer, and Tao Yu. 2022. Unifiedskg: Unifying and multi-tasking structured knowledge grounding with text-to-text language models. *CoRR*, abs/2201.05966.

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. Type-and content-driven synthesis of sql queries from natural language. *arXiv preprint arXiv:1702.01168*.

Xi Ye, Semih Yavuz, Kazuma Hashimoto, Yingbo Zhou, and Caiming Xiong. 2022. Rng-kbqa: Generation augmented iterative ranking for knowledge base question answering. In *ACL*.

Kyra Yee, Yann Dauphin, and Michael Auli. 2019. Simple and effective noisy channel modeling for neural machine translation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5696–5701, Hong Kong, China. Association for Computational Linguistics.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*.

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

Tao Yu, Rui Zhang, He Yang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, et al. 2019. Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases. *arXiv preprint arXiv:1909.05378*.

Liang Zhao, Hexin Cao, and Yunsong Zhao. 2021. GP: Context-free Grammar Pre-training for Text-to-SQL Parsers. *arXiv e-prints*, page arXiv:2101.09901.

A SQL generation analysis

In this section we analyse in detail the predictions generated by T5QL. In Appendix A.1 we measure how often T5QL outputs valid SQLs and give an example of one invalid SQL. In Appendix A.2 we show an example of how constraining column name generation can boost performance.

A.1 Valid SQL generation

First, we check if T5QL using constrained decoding can still generate unparsable SQL. We obtain T5QL-Base's predictions in Spider's development set for beam sizes of 1, 2, 4 and 8. We observe that:

- T5QL never generates an unparsable SQL for the top-1 beam when the beam size > 1.
- Invalid SQL is generated when the LM (i.e., T5) enters a loop, as can be seen in Listing 1. Since the SQL length is limited, T5QL outputs the incomplete (and invalid) SQL. The loop, even if abnormal, is valid SQL syntax, e.g, an average of averages.
- For larger beam sizes (e.g, 8) we saw that the aforementioned model hallucinations are mainly present in the lower scored beams.

Listing 1: Invalid SQL generated by T5QL. For space concerns we abbreviate the generated SQL.

```
from stadium select name, capacity
    order by avg( avg( avg( avg(
    avg( avg( avg( avg( avg( avg(
    avg( avg( avg( avg( avg( avg(
    avg( avg( avg( avg( max( avg(
    min( min( min( min( min( max(
    max( max( max( max( max( max(
    max( max( max( max( max( max(
    max( max( max( max( max( max(
    max( max( max( max( ...
```

Next, we analyse whether model size reduces the number of invalid SQL generated by T5QL. We obtain the predictions in Spider's development set using T5QL-Base and T5QL-Large with and without constrained decoding. We report results of the four methods using 4 beams.

We observe that increasing the size of the model also increases the ability of the model to generate parsable SQL: T5QL-Base wo/ CD generates $\approx 20\%$ invalid SQLs, while T5QL-Large wo/ CD generates only $\approx 5\%$ invalid SQLs (Figure 4). Notice, however, that 5% is still a substantial amount of invalid SQLs. On the other hand, when using constrained decoding, T5QL always produces valid

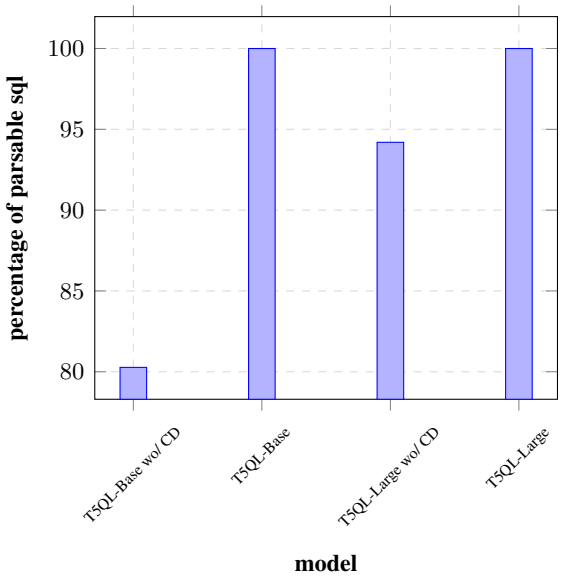

Figure 4: Percentage of parsable SQL, in the Spider's development set, in each model configuration. All methods use beam search with 4 beams and we report results for the first beam.

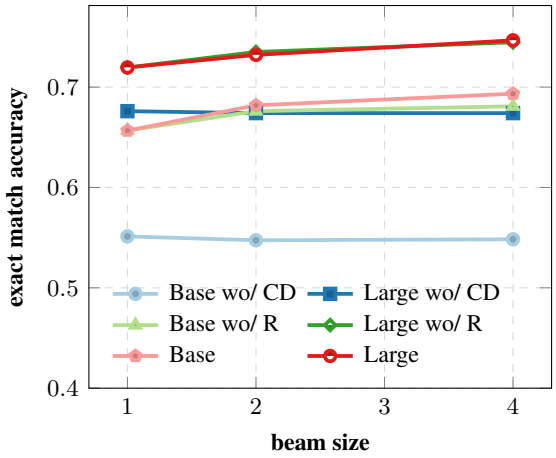

Figure 5: Comparison of EM accuracy in Spider's development between different model configurations. The "Base" model refers to T5QL-Base with constraint decoding and reranking; models "wo/ CD" are the models without constraint decoding nor reranking, whereas models "wo/ R" are the models without reranking.

SQLs when considering only the top-1 beam of beam search with 4 beams; this is true for both T5QL-Base and T5QL-Large.

A.2 Enforce existing table and column names

Finally, we analyse what is the impact of constraining the table and column names during SQL generation. When T5QL does not constrain column and table names, it can generate examples such as the one in Listing 2 where "song_id" is a column name that does not exist in the schema. When constraining column and table names, T5QL always generates existing column and table names, and, in this case, predicts the correct SQL (Listing 3).

Listing 2: Invalid SQL generated by T5QL wo/ CD. In this case the T5QL generated an non-existing column.

```
from singer as t1 join
    singer_in_concert as t2 on
    t1.song_id = t2.song_id
    select t1.name, count( * )
    group by t1.song_id
```

Listing 3: Valid and correct SQL generated by T5QL with CD for the same example as Listing 2.

```
from singer as t1 join
    singer_in_concert as t2 on
    t1.singer_id = t2.singer_id
     select t1.name, count( * )
     group by t1.singer_id
```

B Larger models

Experiments shown for our proposed method, T5QL, used T5-Base as the generator LM. We make this choice since our focus is to show that small LMs can have good performance even when compared against very large LMs. Nevertheless, evaluating if the proposed techniques, namely constrained decoding and reranking, scale to larger LMs is an interesting research question. Thus, we evaluate whether constrained decoding and reranking improve the performance of T5SQL-Large.

From Figure 5 we observe that the performance of T5QL-Base (i.e., Base) is superior to T5-Large (i.e., Large wo/CD) for 2–4 beams. When we add the constrained decoding component to T5-Large (i.e., Large wo/ R), the performance is significantly superior. This results highlights the importance of adding constrained decoding for SQL generation. However, we do not observe gains of adding the reranker model to T5-Large (i.e., Large), which we observed in T5-Base. This might indicate, as we pointed out in Section 3.3, that finding better reranking strategies is an interesting research path.

We do not include results for T5QL-3B since our main goal in this work is to increase performance using multiple smaller components and domain-aware techniques (e.g., constrained decoding) instead of relying on very large models. Furthermore, computing results for T5-3B is very costly in terms of money and time.