Enhancing SQL Query Generation with Neurosymbolic Reasoning

Henrijs Princis[1], **Cristina David**[2], **Alan Mycroft**[1]

[1]University of Cambridge, Cambridge CB3 0FD, UK
[2]University of Bristol, Bristol BS8 1QU, UK
princish@cardiff.ac.uk, alan.mycroft@cl.cam.ac.uk, cristina.david@bristol.ac.uk

Abstract

We propose a neurosymbolic architecture aimed at boosting the performance of any Language Model (LM) for SQL query generation. This approach leverages symbolic reasoning to guide the LM's exploration of the search space by considering multiple paths, symbolically evaluating choices at each decision point to choose the next step, with the added novel ability to backtrack. A key innovation is the use of symbolic checks on both incomplete and fully generated SQL queries, enabling early truncation of unsuccessful search paths. Input consists of textual requirements on the desired query, along with optional example tuples to be selected by the query. Experiments on Xander, our open-source implementation, show it both reduces runtime and increases accuracy of the generated SQL. A specific result is an LM using Xander outperforming a four-times-larger LM.

Code — https://github.com/henrijsprincis/Xander
Spider dataset —
https://huggingface.co/datasets/xlangai/spider
Extended version — https://arxiv.org/pdf/2408.13888

Introduction

Language models (LMs)[1] in particular *large* language models (LLMs) have recently been successfully applied to a wide range of tasks including code generation (Li et al. 2022; Chen et al. 2021; Wang et al. 2021; Le et al. 2022). While initially LMs were used as black-box, monolithic entities, recently there has been a shift towards architectures that allow some form of logical reasoning. For example, the widely-used Chain of Thought (CoT) approach (Wei et al. 2022) enables logical reasoning by breaking down complex problems into smaller steps represented as intermediate thoughts.

However, LMs are still predominantly constrained to *linear search* during inference. For example, in the case of CoT, at each step, the LM commits to a single path, sequentially constructing a solution while disregarding alternative candidates. This linear search strategy has been shown to impede long-term proof planning (Saparov and He 2023; Bubeck et al. 2023), especially in scenarios where multiple valid deduction steps exist. Even when multiple paths are considered (Yao et al. 2023), decisions are still based on the LM self-evaluating its choices.

In this work, we propose an architecture for SQL query generation that leverages symbolic reasoning to enable the *nonlinear exploration of the search space*. Our approach navigates a solution tree using Best-First Search (Pearl 1984), with support for backtracking. This architecture is compatible with any pretrained LM, and we apply it to several smaller, open-source LMs, where we show that it improves their performance out-of-the-box. While our focus is on SQL, our broader goal is to illustrate that a neurosymbolic approach provides an alternative to scaling model size. This addresses the high computational cost of LMs, where accuracy hinges heavily on the parameter count (Kaplan et al. 2020).

Our architecture explores the search space guided by a symbolic query checker, a neural query checker, and a symbolic test and repair module. When integrated with a generic pretrained LM, these modules prune bad queries before they have finished being generated and correct errors made by the LM. As far as we are aware LM-based nonlinear solution-building has not been investigated in the context of code generation, where the solution space is particularly large.

One challenge when generating code is that the result can only be checked for correctness (i.e. adherence to the original intent) once it is completely generated— here defined as either including a semicolon or reaching an end-of-sequence token predicted by the language model. In order to address this, one of the innovations of our work is that the symbolic query checker is designed so that it can test incomplete queries. The ability to exclude erroneous candidates as early as possible is critical when exploring a vast solution space.

As an additional challenge, SQL's flexibility in expressing the same query in multiple ways can hinder the LM's performance, as the model is penalised for predicting logically equivalent but syntactically different queries. To address this, we adapt NLP techniques (Tabassum and Patil 2020; Chai 2023) to eliminate unnecessary information from SQL queries before training an LM. In particular, we propose Standardised SQL, a form of SQL that minimises stylistic variations. Standardised SQL enhances both LM fine-tuning

[1]In this work, we use the term language model (LM) to refer to a language model of any size.

and inference, where it can be easily verified by symbolic modules. Our results show that Standardised SQL significantly improves overall LM accuracy.

Query synthesis The task of query synthesis has been previously considered in literature either as *text-to-SQL* or *Query-by-Example (QBE)*: text-to-SQL denotes the task of generating SQL code from a textual description (Wang et al. 2022; Herzig and Berant 2018; Seneff et al. 1991; Li et al. 2023a; Wang et al. 2019; Qi et al. 2022; Scholak, Schucher, and Bahdanau 2021), whereas QBE aims to find a query that returns user-provided output examples (Lissandrini et al. 2018; Tran, Chan, and Parthasarathy 2014; Psallidas et al. 2015; Shen et al. 2014; Tran, Chan, and Parthasarathy 2014; Wang, Cheung, and Bodik 2017).

In this work, we focus on the combined task: generating queries from natural language descriptions optionally accompanied by output examples. Our decision to include examples was inspired by tools like Excel's FlashFill (Menon et al. 2013), a Programming-by-Example (PbE) approach, where users provide a few examples and Excel infers the pattern to complete the data. From the point of usability, examples are an easy way for novice users to clarify their intent, a core motivation in the area of PbE. Accompanying natural language descriptions with input-output examples is commonly used for code generation in general-purpose languages (Li et al. 2022; Le et al. 2022; Jiang et al. 2024), it has been far less studied for SQL—the only work we are aware of is (Baik et al. 2020).

While approaches to text-to-SQL mostly rely on custom neural architectures, QBE techniques typically use symbolic reasoning. We explore a unified approach where, instead of designing a custom neural model, we propose a neurosymbolic design that can be used to enhance any pretrained LM without modifications.

Problem Definition We consider the task of generating SQL code from a textual description and zero or more user-provided *output examples*. We call this task *text-to-SQL-with-examples* and it can be defined as follows. Given a natural language question q, the database $\mathcal{D}$ which has a schema $\mathcal{D}_{schema}$, and possibly an empty set of output examples o, we wish to find an SQL query l which when executed on database $\mathcal{D}$ answers question q by returning a set s of tuples such that $s \supseteq o$. (This "open-world" formulation is appropriate as we cannot expect the user to provide an exhaustive set of output examples o.)

Contributions
1. We propose a neurosymbolic design for generating SQL queries from natural language and examples; our design can be used to enhance the performance of any pretrained LM without modifications. This approach incorporates symbolic reasoning to guide the LM's solution search by exploring multiple paths, symbolically evaluating choices at each decision point to choose the next step, with the added ability to backtrack. This advancement is particularly significant because LMs typically commit to a single path when presented with multiple valid deduction options, lacking the systematic exploration of alternatives.

Algorithm 1: SQL Query Generation with Xander

```
Require: D: Database, q: question, o: output examples,
    BPE: Byte-pair encoding, BPD: Byte-pair decoding,
    LM: Language Model, PQC: Partial-Query Checker,
    QTR: Query Test-and-Repair
 1: $\mathbf{x} \leftarrow \mathbf{BPE}(D_{\text{schema}}, q, o)$    ▷ The byte pair encoding of Database schema, question and output examples is used as input.
 2: priority_queue.setempty();
 3: $l, p_l \leftarrow$ "", 1
 4: while True do
 5:     if is_complete_query($l$) then
 6:         $l \leftarrow \mathbf{QTR}(l, D)$    ▷ Test and Attempt Repair
 7:         if QTR succeeds then
 8:             return $l$
 9:         end if
10:     else
11:         $\mathbf{y} \leftarrow \mathbf{BPE}(l)$
12:         $l\_continuations \leftarrow \mathbf{LM}(\mathbf{x}, \mathbf{y})$    ▷ Predict next tokens with probabilities
13:         for $t, p_t$ in $l\_continuations$ do
14:             $l' \leftarrow l + \mathbf{BPD}(t)$    ▷ Proposed new query
15:             $p'_l \leftarrow p_l \times p_t$    ▷ Proposed probability
16:             if $\mathbf{PQC}(l', o, D_{schema})$ then
17:                 priority_queue.add($(l', p'_l)$)
18:             end if
19:         end for
20:     end if
21:     $l, p_l \leftarrow$ priority_queue.pop() ▷ Pop next $\ell$ to explore and its corresponding probability.
22: end while
```

2. We investigate the use of normalisation in code generation by introducing *Standardised SQL*.
3. We implemented a prototype tool called Xander, and showed it leads to significant improvements in runtime and accuracy, offering a compelling alternative to scaling model size.

General Architecture of Xander

In this section, we present the design for our neurosymbolic technique, and its corresponding implementation Xander. A high-level overview of Xander during inference is given in Figure 1, and a detailed description is captured in Algorithm 1. Xander takes as input a 3-tuple $(\mathcal{D}, q, o)$ consisting of database $\mathcal{D}$, natural language question q, and output-example set o. We then use $(\mathcal{D}_{schema}, q, o)$, where $\mathcal{D}_{schema}$ is the schema of D, to prompt the LM.

In Algorithm 1, $\mathbf{x}$ denotes the task description consisting of the concatenation of database schema $\mathcal{D}_{schema}$, question q and examples o, and $\mathbf{y}$ represents the possibly empty partial query that will fulfill the task description once the query is fully generated.

We use Byte-Pair Encoding (BPE) (Gage 1994) to encode the task description and partial query as sequences of tokens meaning that one can think of $\mathbf{x}$ and $\mathbf{y}$ at lines 1 and 11 as: $\mathbf{x} = [x^1, x^2, ..., x^n]$ and $\mathbf{y} = [y^1, y^2, ..., y^{m-1}]$, respectively.

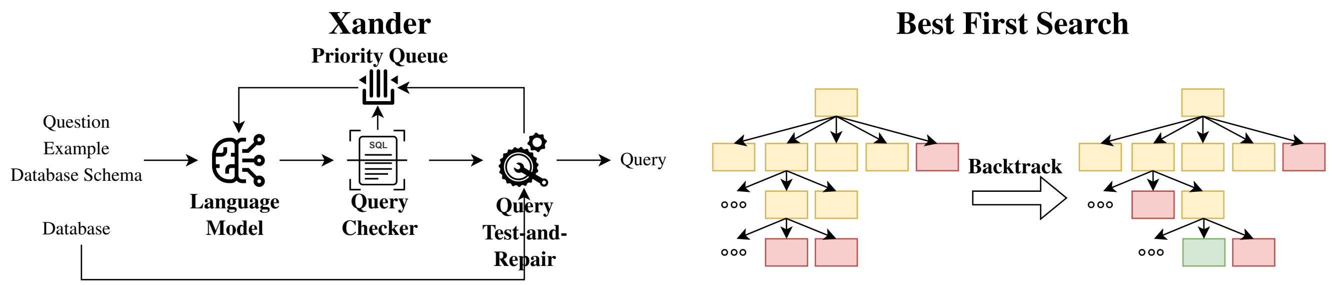

Figure 1: During inference Xander consists of three main modules connected by Best First Search (BFS). BFS explores the solution tree by choosing the most promising incomplete query (yellow box) and passing it to the LM. The LM then provides five candidates that extend the query by a single token. A Partial-Query Checker dismisses candidate queries containing errors (red box). The Partial-Query Checker passes queries without errors back to BFS if they are still incomplete or to Query Test-and-Repair if they are complete. Finally, Query Test-and-Repair module executes the full query and verifies that it yields the correct result, in which case it is returned to the user (green box). If it does not, they are dismissed (red box) and BFS backtracks.

In each iteration of the while loop starting at line 4, we explore the current query ℓ (initially the empty string). We first look at the scenario where query ℓ is not yet complete, meaning that we have to continue generating tokens for it. This is captured starting with line 10 in Algorithm 1. After encoding ℓ into $\mathbf{y}$, we call the LM to provide us the next token at line 12. Generally speaking, the LM defines a probability distribution p over the possible continuations of $\mathbf{y}$, where the most likely next token y^m in the partial query is obtained with the following equation:

$$y^m = \arg\max_t p(t|\mathbf{x}, \mathbf{y})$$

In Algorithm 1, the call to the LM returns the most promising five pairs of candidate next tokens and their respective probabilities, which get stored in $l\_continuations$.

Each newly obtained query ℓ' (the concatenation of the previous incomplete query ℓ and the decoding of a newly generated token t) are checked for plausibility by the Partial-Query Checker **PQC** (line 16 in Algorithm 1). As we explain later, the default **PQC** module applies symbolic checks to detect invalid queries. Given that the LM generates queries one word at a time, completely generating a query and then checking it for errors is computationally wasteful. Instead, Xander checks incomplete queries, thus discarding many invalid ones early on.

The queries that pass the **PQC** are added to $priority\_queue$. This priority queue allows us to avoid generating the same query twice, as well as to use Best First Search (BFS) in order to explore the space of candidate queries. To enable the latter, the priority queue is ordered by the probability of each individual query. Then, BFS explores the solution tree by expanding the most promising node, which corresponds to an incomplete query with the highest probability. For this purpose, in each iteration of the while loop at line 4, we pop an incomplete query with the highest probability from the priority queue and further explore it (line 21 in Algorithm 1).

Intuitively, the priority queue corresponds to a tree of possible candidates as shown on the RHS of Figure 1, where each node represents a query (while some leaves may be complete queries, the rest are incomplete), and we can think of edges as corresponding to a token that concatenated to the parent query produces the child query. Notably, we can have cases where, for a given incomplete query, all the generated next tokens result in queries that have lower probability than some of the queries already in the priority queue. Consequently, the query that is popped (in order to be explored in the next iteration of the while loop) is an ancestor of the current query. This corresponds to backtracking in the tree of candidate queries.

If we find a complete query, it is checked by Query Test-and-Repair **QTR** (line 6 in Algorithm 1). Essentially, the query is executed on the full database D to give a tuple set s. If $s \supseteq o$, then the corresponding complete query is reported to the user as the proposed query. Otherwise the **QTR** module tries to repair ℓ using fuzzed variants of ℓ that are at a Hamming distance of one. We regard any query that differs by exactly one aggregation operation (e.g. AVG), conditional operation (AND), comparison operator ($\geq$) or table column combination ($t1.c1$) as satisfying the requirement. Each variant is executed and its tuple set compared with o as before.

The output of Xander is Standardised SQL—a modified form of SQL that enforces stricter syntax and a consistent style. Standardised SQL increases accuracy and reduces the difficulty of checking whether a partial query contains an error (more details in the following section).

Standardised SQL

In this section, we address the limitations of SQL in the context of LMs and introduce Standardised SQL as a solution.

SQL allows multiple ways to express the same query, leading to inconsistencies in training datasets that can hurt LM performance. During training, LMs may be penalised for predicting a logically correct but stylistically different query, causing unnecessary penalties due to variations in query style (e.g. capitalisation differences). This inconsistency acts like random noise during training, similar to issues encountered in natural language processing (NLP).

To mitigate this, NLP techniques suggest pre-processing

SQL

```
SELECT avg(t1.age) ,   min(t1.age)
FROM singer AS t1 WHERE
t1.country = "France"
```

STANDARDISED SQL

```
SELECT avg( singer.age ), min( singer.age )
FROM singer
WHERE singer.country = "FRANCE"
GROUP BY NONE
ORDER BY NONE
HAVING NONE
LIMIT NONE
INTERSECT NONE
EXCEPT NONE
UNION NONE;
```

Figure 2: SQL (left) Standardised SQL (right). SQL uses an alias which is highlighted in red. Standardised SQL replaces the alias with what it points to (t1 is replaced with singer).

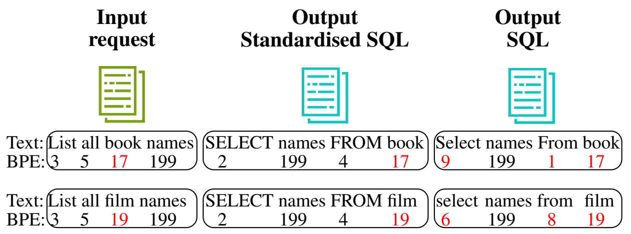

Figure 3: Illustration of how Standardised SQL makes learning easier. Note how, in the (unstandardised) SQL case, two different annotation styles such as capitalising SQL keywords can cause two queries with similar input requests to have drastically different output byte-pair encodings.

data to eliminate unnecessary variations before training (Tabassum and Patil 2020; Chai 2023). Inspired by these methods, we developed Standardised SQL—a stricter syntax that normalises SQL queries to reduce stylistic variations. Standardised SQL adheres to the following rules: **(1)** SQL keywords are uppercase, following standard typing conventions; **(2)** Keywords are followed by a space, which makes vocabulary checking easier; **(3)** All clauses are included and non-optional (this is to eliminate the need for the LM to learn the clause order, e.g. a query where FROM precedes SELECT will error); **(4)** All aliases are expanded (during testing, we discovered that LMs frequently struggle to determine the association between columns and tables when aliases are used in queries).

An example of an SQL query and its corresponding Standardised SQL is given in Figure 2. Then, in Figure 3, we have two natural language requests: "List all book names" and "List all film names", and their corresponding SQL and Standardised SQL output queries. The figure also contains the byte-pair encodings. For Standardised SQL, the difference in the output's encoding between the two queries is considerably smaller than for SQL. The variation in SQL comes down to low-level stylistic choices chosen by different annotators which makes the BPE output look artificially more different than it is.

One of the benefits of Standardised SQL is that it enables the checking of partial queries by the Partial-Query Checker.

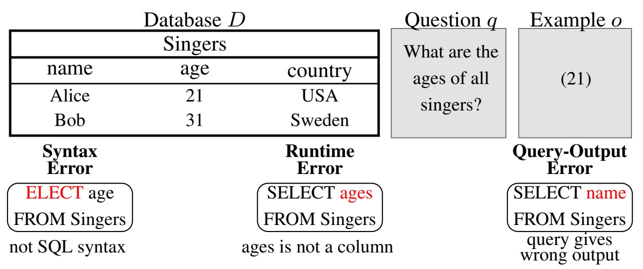

Figure 4: Illustration of three types of errors. The sample SQL marked "query-output error" is syntactically correct and executes, but its returned tuples do not contain the example tuple provided by the user.

Intuitively, the stricter syntax of Standardised SQL makes it easier to find faulty queries early on.

The Partial-Query Checker is an important part of Xander's architecture. It takes as input the user request (i.e. the question in natural language and any output examples), the database schema and a partial SQL query $\mathbf{y}$, and investigates whether the query is valid (i.e. it can be completed such that the final query corresponds to the initial question) or invalid (i.e. there is no way of completing it in a way that corresponds to the initial question). For this purpose, the Partial-Query Checker checks the query for common *syntax*, *runtime*, and *query-output errors* (see Figure 4 for instances of such errors). A *query-output error* occurs when the generated query does not return all the output examples provided by the user in the user request. At first it is surprising that query-output errors can be detected without running the query on the database, but sometimes (often enough) query-output errors can be detected just given the database schema—using techniques described below.

As aforementioned, the Partial-Query Checker only has access to the database schema and not the full database (by contrast the Query Test-and-Repair module described later tests queries on the full database). As part of our evaluation, the Partial-Query Checker is pluggable—we have a standard Partial-Query Checker, the *Symbolic Query Checker*, but also a neural alternative, the *Neural Query Checker*. Both are explained below.

Symbolic Query Checker In order to detect the aforementioned errors, the **PQC** performs three types of checks: vocabulary, scope and type checking.

Vocabulary Checking The Standardised SQL query is checked clause by clause. Every clause is checked for containing the right vocabulary. For example, the **SELECT** clause can only contain aggregation operations, brackets and table-column names. In this step we also reject table-column combinations which do not belong to the schema $\mathcal{D}_{schema}$. Most errors caught by vocabulary checking will be syntax errors; however, it will also catch the runtime error of selecting an invalid table-column combination.

Scope Checking We check that all of the tables referred to by **SELECT**, **WHERE**, **GROUPBY**, and **HAVING**, clauses also appear in the **FROM** clause. Scope checking can only detect runtime errors.

Type Checking Finally, the types of the tuples which are returned by the **SELECT** clause are compared to the type of the tuple(s) provided as examples. This ensures that the right number of columns with the right types are chosen. Type checking only catches query-output errors. For illustration, in the example in Figure 4, the query illustrating "query-output error" returns a string, whereas the expected output is an integer.

Neural Query Checker We explored an alternative plug-in to the Symbolic Query Checker and conducted an experiment comparing it to the Neural Query Checker. While the Symbolic Query Checker proved more accurate and was used in our experiments, we also discuss the Neural Query Checker for completeness.

The Neural Query Checker is a neural network designed to classify whether a partial query is correct or contains an error, functioning as a *critic* in reinforcement learning. The main challenge is identifying the location of errors within the query. To address this, we use a method from CodeRL (Le et al. 2022), where the Neural Query Checker processes the query word by word, generating hidden states that encode information about potential mistakes in the partial query. These hidden states are then aggregated using *max pooling*, producing a single representation that determines whether the entire query contains an error (see Figure 5).

To train the Neural Query Checker, we generate queries using the training dataset, one of the fine-tuned LMs, and Beam Search (Freitag and Al-Onaizan 2017), labelling each query based on its execution result. Although training uses only complete queries, we expect the network's accuracy on incomplete queries to be similar due to the max pooling layer, which helps identify mistakes as they occur.

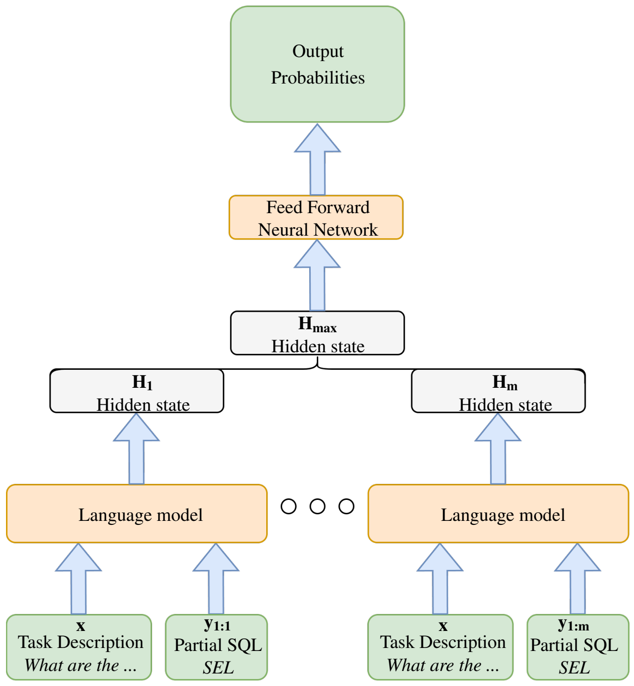
Output Probabilities
Feed Forward Neural Network
$\mathbf{H_{max}}$ Hidden state
$\mathbf{H_1}$ Hidden state
$\mathbf{H_m}$ Hidden state
Language model
Language model
$\mathbf{x}$ Task Description What are the ...
$\mathbf{y_{1:1}}$ Partial SQL SEL
$\mathbf{x}$ Task Description What are the ...
$\mathbf{y_{1:m}}$ Partial SQL SEL

Figure 5: Illustration of the Neural Query Checker. $\mathbf{y_{1:i}}$ denotes the subsequence of the tokenised partial Standardised SQL query generated by a language model up to token i. The length of the partial query is m. A hidden state $\mathbf{H_i}$ is produced for every subsequence starting at position 1. This hidden state $\mathbf{H_i}$ is then used as input to max pooling which works across sequence length. The result is input into another Feed Forward Neural Network and finally Softmax is applied to generate the output probabilities which correspond to "correct", "query-output error", "runtime error", and "syntax error".

Query Test-and-Repair

Given a complete query, after we checked it with the Partial-Query Checker, we execute it to verify that it produces a subset of the user-provided output examples. If it does not, the Query Test-and-Repair module tries to repair it. An important design consideration of the Query Test-and-Repair module is that it should keep the (Hamming) difference between the repaired query and the generated query to a minimum. If the repaired query diverges significantly from the original, it may indicate a flawed correction that overlooks the user's language specification, resulting in a potential false positive.

In this work the Query Test-and-Repair module generates all queries that differ from the original by exactly one *enumerative* SQL token (these are SQL operations, column names, or table names). An example is given in Figure 6. It does not try to repair off-by-one errors for constants (e.g. by changing "France" to "Grance") due to very large search-space and low likelihood of success.

Experimental Setup

Environment Experiments used Python with the Hugging Face transformers library (Wolf et al. 2020). Except those for Microsoft Phi-1.5, experiments were performed using Tesla P100 GPU and Intel Xeon 6142 CPU. For Microsoft Phi-1.5, due to the larger model size, we used an Amazon EC2 G5.xlarge instance with an A10G (24GB) GPU.

LMs We considered the following LMs for which weights are publicly available: CodeT5 (Wang et al. 2021), BART (Lewis et al. 2019), CodeGen (Nijkamp et al. 2023), and Microsoft Phi-1.5 (Li et al. 2023b). As a comparison point, we used GPT-3.5 Turbo, GPT-4o mini, and GPT-4o, which are not open-source.

Dataset We used the Spider (Yu et al. 2018) dataset, which is the most challenging benchmark for the cross-domain and multi-table text-to-SQL. The training set was used to finetune the network and the validation set was used to measure real-world accuracy and runtime.

Since the Spider dataset does not include output examples in the user description, we generate them by executing the golden queries and taking the first returned tuple. As we work with Standardised SQL, we rewrote the Spider dataset to follow Standardised SQL's constraints. There are a few

SELECT avg (singer.age)
FROM singer
WHERE singer.country = “France”

Tables	Columns	Aggregation	Comparison Op		
singer	singer.country	avg	=	>=	<=
dancer	singer.age	min	!=	>	<
concert	singer.sname	max			

Figure 6: The **QTR** module takes in a given query (top) and generates all queries of Hamming distance 1, i.e. they differ by one table name (blue), column name (green), aggregation (red), or comparison operator (purple). One such generated query is **SELECT** *max*(singer.age) **FROM** singer **WHERE** singer.country = “France”; this differs from the given query by using “max” instead of “avg”.

queries in the original Spider dataset that use complex joins and, therefore, cannot be expressed in Standardised SQL. After removing those, we have 6779 queries in the training dataset, and 1018 in the validation dataset (compared to 7000 and 1034, respectively, in the original Spider dataset).

Training Throughout our experiments we use pretrained LMs. These LMs are further trained (finetuned) to generate SQL or Standardised SQL on the Spider dataset or its Standardised SQL version, depending on the experiment. All networks except Microsoft Phi-1.5 were fitted for 50 epochs with batch size of 10. The Adam (Kingma and Ba 2017) optimiser with a learning rate of $4e^{-5}$ was used to find the optimal weights. For Microsoft Phi-1.5, to save memory, batch-size of 1 was used and RMSProp optimiser was used instead of Adam. To account for the larger network size, the learning rate was reduced to 4e^{-6} and the network was fitted for 5 epochs.

Encoding The input to the transformer was the byte-pair encoding BPE of the serialised concatenation of the user question, database schema, and output examples, i.e.

$$BPE(Concat(str(q), str(\mathcal{D}_{schema}), str(o)), 32100)$$

where 32100 is the size of the vocabulary. The database contents were not included in the specification because doing so would require too much memory. The golden output was the tokenisation of SQL or Standardised SQL depending on the experiment, both for training and validation datasets.

Query Evaluation Queries were assessed using Distilled Test Suites (Zhong, Yu, and Klein 2020), leveraging two key metrics: *exact-match accuracy* and *execution accuracy*. *Exact-match accuracy* determines whether the abstract syntax tree (AST) of the generated query aligns perfectly with that of the reference query contained in the Spider dataset, excluding constants. *Execution accuracy*, evaluates whether the generated query retrieves the correct tuples when executed on a predefined database instance. The reference query serves as the “gold standard,” representing the ideal output, while the generated query is tested against it to measure how closely the language model’s output adheres to the expected behaviour.

Experimental Results

Generalisability

The aim of this experiment is to explore whether a neurosymbolic approach is faster in terms of runtime and accuracy compared to a purely neural approach. The secondary goal of the experiment is to investigate whether a neurosymbolic approach could provide similar accuracy benefits as scaling the model size.

In all instances, we provide both the natural language description and the user-provided output tuple, and allow a one-minute time limit to generate a query which returns the user-provided output tuple when executed. In order to enable a fair comparison with Xander, we also enable search of the solution space for the approach without Xander, which also constructs the solution by querying the LM for one token at a time and building a solution tree that gets explored using BFS. Essentially, it follows the same algorithm as the one for Xander in Algorithm 1, with the only exceptions that, at line 6, it only tests the complete query without attempting repair, and it does not call the Partial-Query Checker at line 16. Instead, all incomplete queries are added to the priority queue.

For the GPT models, due to monetary limitations, we accessed them (through OpenAI’s API) using a single query, which may explain their poorer performance. This allowed us to explore whether smaller open-source LMs enhanced with Xander, can rival much larger closed-source models used off-the-shelf.

The results of this experiment can be seen in Table 1. The main conclusions are:

1. Xander increases execution accuracy by an average of 10.9%, exact-match accuracy by 6.0% and it finds the correct query 28% faster than using a purely neural approach in the form of a LM. Xander always improves execution accuracy, but we note that for BART model it decreased exact-match accuracy. This is likely because exact-match accuracy has a high false-negative rate.

2. A neurosymbolic approach offers a compelling alternative to scaling model size. **The CodeT5 small model *beats* the four-times-larger CodeT5 base model when CodeT5 small is used with Xander.** Also on this point, all the GPT models performed worse than up to an order of magnitude smaller fine-tuned LMs with Xander.

Xander Ablation

For the ablation experiment we keep the same experimental setup from the generalisability experiment and only consider the CodeT5 small and CodeT5 base models.

The aim of the ablation experiment is to investigate what the most important factors for determining Xander’s performance are. To this end, we measure the accuracy and runtime benefits of Standardised SQL, Partial-Query Checker, Query Test-and-Repair, and whether output examples are provided as a part of the user request. The experimental results are reported in Table 2. Training configurations are in bold. The word “examples” in training configuration refers to whether or not user-provided output examples were included in the task description. In normal font, we have the

Model	#parameters	Training Time	Validation Time per sample (s)	Exact-Match Accuracy (%)	Execution Accuracy (%)
CodeT5 small	(62M)	3h8m	15.7	60.7	69.9
Ditto with Xander		2h59m	9.1	66.0	78.6
CodeT5 base	(222M)	10h8m	10.7	63.5	73.6
Ditto with Xander		9h39m	7.3	68.9	80.5
BART	(139M)	6h8m	25.3	51.5	58.1
Ditto with Xander		5h50m	19.5	39.0	59.0
CodeGen	(350M)	19h22m	54.6	4.4	5.9
Ditto with Xander		18h31m	47.5	6.8	14.4
Microsoft Phi-1.5	(1.3B)	3h49m	34.1	30.5	42.3
Ditto with Xander		3h41m	18.1	59.8	72.0
GPT-3.5 Turbo	(Unknown)	N/A	0.9	33.5	69.1
GPT-4o mini	(Unknown)	N/A	1.0	30.0	75.2
GPT-4o	(Unknown)	N/A	1.3	38.8	76.1

Table 1: Table showcasing the results of various LMs with and without using Xander. High accuracy, low training and generation times are desirable. The best performing model is highlighted.

settings used for inference. The entry "1 attempt" means that we stop generating queries after we find a single complete query. The entry "multiple attempts" means we repeatedly generate queries, stopping when we find one that generates the user-provided output examples (or we reach one-minute time limit). The entry "PQC" refers to including the Partial-Query Checker and the entry "repair" refers to including Enumerative Repair in the Query Test-and-Repair module.

The most important factor in determining Xander's accuracy is Standardised SQL which provides a 9.5% execution accuracy increase when a single attempt is allowed. The addition of the Partial-Query Checker gives further 4.2% improvement to execution accuracy. Using output examples as part of the description, only improves exaction accuracy by 0.3%. Finally, the Query Test-and-Repair module improves exaction accuracy only by 0.2%, but improves the time to find a query by an average of 25%.

Partial-Query Checker Comparison Experiment

Queries from the validation set were used to compare the efficacy of Neural Query Checker and Symbolic Query Checker. In particular, we generate top four most promising queries for each validation question. Then, we compare the classifications obtained by using the Symbolic Query Checker with the classification obtained by using the Neural Query Checker. Finally, we construct two confusion matrices to compare their accuracy (see Figure 7). The results allow us to draw the following conclusions:

1. The Symbolic Query Checker is 22% more accurate than the Neural Query Checker at recognising when a query contains a mistake. This can be at least partly attributed to the Neural Query Checker's inability to recognise runtime errors which occurred in 28% of the queries.

2. The Symbolic Query Checker cannot differentiate between query-output errors and correct queries. This is because only column types are used to detect query-output errors and column type mistakes are rarely (1.3% of all queries) made by the LM.

3. The Neural Query Checker can detect most syntax errors. While this is a promising finding, it is worth noting that the LM also rarely makes syntax errors. Indeed, only 5% of the queries contain a syntax error.

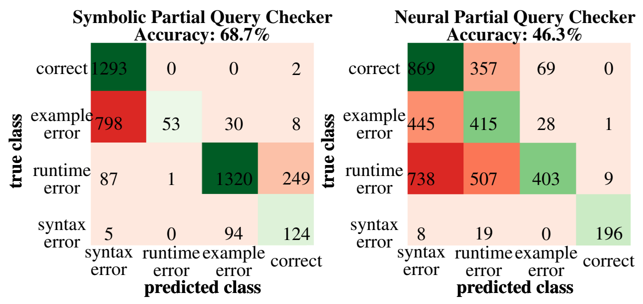

Figure 7: Confusion Matrix for Symbolic Query Checker (left) and Neural Query Checker (right). High values in the diagonal (green) are good, high values off diagonal (red) are bad. The rows of the confusion matrix denote the true label obtained by executing the query. An ideal Partial-Query Checker would only have elements on the diagonal. By summing the rows, we see that most common type of mistake was a runtime error. The columns represent the predicted class. The elements in the diagonal represent how many times the predicted class matched up with the true class (higher is better). The Symbolic Query Checker makes mistakes in the error type detection.

Comparison to Existing Works

To the best of our knowledge, the only other work that accepts both NL and output examples is **Duoquest** (Baik et al. 2020). While they do not support 445 (43%) queries of the Spider validation dataset (e.g. clauses with multiple selection predicates), for the supported queries, they report an **execution accuracy of 63.5%**.

From text-to-SQL, **Picard** (Scholak, Schucher, and Bahdanau 2021) is the closest to our approach, as it enhances fine-tuned LMs out-of-the-box by rejecting inadmissible to-

Model	**Training Configuration** Xander options	Training Time	Validation Time per sample (s)	Exact Match Accuracy (%)	Execution Accuracy (%)
CodeT5 small	**SQL + No Examples + No Finetuning**	**0h0m**	-	-	-
	1 attempt	-	30.6	0	0
CodeT5 small*	**SQL + No Examples**	**3h4m**	-	-	-
	1 attempt	-	11.6	1.7	2.4
CodeT5 small	**SQL + No Examples**	**3h4m**	-	-	-
	1 attempt	-	0.5	46.9	48.7
CodeT5 small	**Standardised SQL + No Examples**	**2h59m**	-	-	-
	1 attempt	-	0.9	56.4	58.4
CodeT5 small	**Standardised SQL + Examples**	**2h59m**	-	-	-
	1 attempt	-	0.8	56.8	59.7
	PQC + 1 attempt	-	1.1	59.0	63.9
	multiple attempts	-	12.8	66.8	76.0
	PQC + multiple attempts	-	10.2	68.3	78.4
	repair + PQC + multiple attempts	-	9.1	66.0	78.6
CodeT5 base	**Standardised SQL + Examples**	**9h39m**	-	-	-
	PQC + 1 attempt	-	2.7	62.9	66.5
	PQC + multiple attempts	-	10.3	71.2	80.4
	repair + PQC + multiple attempts	-	7.3	68.9	80.5

Table 2: Table showcasing a Xander ablation study. The "*" denotes that weights were initialised randomly as opposed to using the pretrained weights. The row containing the best accuracy is highlighted.

kens. However, Picard does not use output examples, explores the solution space differently by using beam search instead of best-first search, and does not attempt to repair solutions. In experiments, Picard improves **exact-match accuracy by 5.15% and execution accuracy by 6.5%** over base LMs. These improvements are smaller than those achieved by Xander (6% and 10.9%, respectively). Since Picard does not use output examples, a direct comparison is not possible.

The majority of the works on text-to-SQL require customising existing LMs. For instance, **RASAT** (Qi et al. 2022) augments the self-attention modules in a model's encoder and introduces new parameters to the model. When customising T5, it achieves **exact-match accuracy of 72.6% and execution accuracy of 76.6%** on the Spider validation set. **REDSQL** (Li et al. 2023a) breaks SQL generation into the generation of a skeleton of SQL keywords, which is then filled in with the missing values. For this purpose, it relies on a a ranking-enhanced encoder to alleviate the effort of the schema linking and a skeleton aware decoder to implicitly guide the SQL parsing. By customising T5-base, REDSQL achieves **exact-match accuracy of 71.7% and execution accuracy of 77.9%** on the Spider validation set. Both RASAT and REDSQL achieve marginally better exact-match accuracy but lower execution accuracy. We hypothesise that by customising the base LM, RASAT and REDSQL are able to better capture the syntactic patterns in the Spider dataset which leads to higher exact-match accuracy, while Xander captures the intent of the query by deriving information from the user-provided output examples which increases execution accuracy.

Multi-agent Architectures with LMs While, initially, LMs were used as black-box, monolithic entities, recently, there has been a shift towards architectures that foster some form of logical reasoning as part of the problem-solving process, sometimes by leveraging additional, possibly non-neural systems (Karpas et al. 2022; Creswell and Shanahan 2022; Shen et al. 2023; Wei et al. 2022). Given that LLMs were shown to have difficulty with proof planning when using a linear search strategy (Saparov and He 2023), other works are focused on decision-making and space exploration (Schlag et al. 2023; Liu et al. 2023; Yao et al. 2023; Long 2023). As opposed to these works, we propose a neurosymbolic architecture for generating SQL queries that uses verification and repair modules to guide a non-linear exploration of the solution space.

Conclusions

This work explored whether neurosymbolic approaches could serve as an alternative to scaling model size. Specifically, we built a tool called Xander for the text-to-SQL-with-examples task. Xander is able to dismiss large parts of the search-space by standardising SQL and pruning bad queries before they have finished generating. Evaluation of Xander showed that neurosymbolic approaches can lead to significant improvements in runtime and accuracy, enabling a smaller LM to outperform its four-times-larger counterpart. **This means that neurosymbolic approaches provide a compelling alternative to the de facto industry standard of improving performance by scaling model size.**

Acknowledgements Cristina David was supported by the Royal Society University Research Fellowship URF\R \221031.

References

Baik, C.; Jin, Z.; Cafarella, M. J.; and Jagadish, H. V. 2020. Duoquest: A Dual-Specification System for Expressive SQL Queries. *CoRR*, abs/2003.07438.

Bubeck, S.; Chandrasekaran, V.; Eldan, R.; Gehrke, J.; Horvitz, E.; Kamar, E.; Lee, P.; Lee, Y. T.; Li, Y.; Lundberg, S. M.; Nori, H.; Palangi, H.; Ribeiro, M. T.; and Zhang, Y. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. *CoRR*, abs/2303.12712.

Chai, C. P. 2023. Comparison of text preprocessing methods. *Natural Language Engineering*, 29(3): 509–553.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H. P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; Ray, A.; Puri, R.; Krueger, G.; Petrov, M.; Khlaaf, H.; Sastry, G.; Mishkin, P.; Chan, B.; Gray, S.; Ryder, N.; Pavlov, M.; Power, A.; Kaiser, L.; Bavarian, M.; Winter, C.; Tillet, P.; Such, F. P.; Cummings, D.; Plappert, M.; Chantzis, F.; Barnes, E.; Herbert-Voss, A.; Guss, W. H.; Nichol, A.; Paino, A.; Tezak, N.; Tang, J.; Babuschkin, I.; Balaji, S.; Jain, S.; Saunders, W.; Hesse, C.; Carr, A. N.; Leike, J.; Achiam, J.; Misra, V.; Morikawa, E.; Radford, A.; Knight, M.; Brundage, M.; Murati, M.; Mayer, K.; Welinder, P.; McGrew, B.; Amodei, D.; McCandlish, S.; Sutskever, I.; and Zaremba, W. 2021. Evaluating Large Language Models Trained on Code. *CoRR*, abs/2107.03374.

Creswell, A.; and Shanahan, M. 2022. Faithful Reasoning Using Large Language Models. *CoRR*, abs/2208.14271.

Freitag, M.; and Al-Onaizan, Y. 2017. Beam Search Strategies for Neural Machine Translation. *CoRR*, abs/1702.01806.

Gage, P. 1994. A New Algorithm for Data Compression. *C Users J.*, 12(2): 23–38.

Herzig, J.; and Berant, J. 2018. Decoupling Structure and Lexicon for Zero-Shot Semantic Parsing. *CoRR*, abs/1804.07918.

Jiang, J.; Wang, F.; Shen, J.; Kim, S.; and Kim, S. 2024. A Survey on Large Language Models for Code Generation. *CoRR*, abs/2406.00515.

Kaplan, J.; McCandlish, S.; Henighan, T.; Brown, T. B.; Chess, B.; Child, R.; Gray, S.; Radford, A.; Wu, J.; and Amodei, D. 2020. Scaling Laws for Neural Language Models. *CoRR*, abs/2001.08361.

Karpas, E.; Abend, O.; Belinkov, Y.; Lenz, B.; Lieber, O.; Ratner, N.; Shoham, Y.; Bata, H.; Levine, Y.; Leyton-Brown, K.; Muhlgay, D.; Rozen, N.; Schwartz, E.; Shachaf, G.; Shalev-Shwartz, S.; Shashua, A.; and Tennenholtz, M. 2022. MRKL Systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning. *CoRR*, abs/2205.00445.

Kingma, D. P.; and Ba, J. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980.

Le, H.; Wang, Y.; Gotmare, A. D.; Savarese, S.; and Hoi, S. C. H. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning.

Lewis, M.; Liu, Y.; Goyal, N.; Ghazvininejad, M.; Mohamed, A.; Levy, O.; Stoyanov, V.; and Zettlemoyer, L. 2019. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. *CoRR*, abs/1910.13461.

Li, H.; Zhang, J.; Li, C.; and Chen, H. 2023a. RESDSQL: Decoupling Schema Linking and Skeleton Parsing for Text-to-SQL. arXiv:2302.05965.

Li, Y.; Bubeck, S.; Eldan, R.; Giorno, A. D.; Gunasekar, S.; and Lee, Y. T. 2023b. Textbooks Are All You Need II: phi-1.5 technical report. This is a technical report from Microsoft Research.

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Lago, A. D.; Hubert, T.; Choy, P.; de Masson d'Autume, C.; Babuschkin, I.; Chen, X.; Huang, P.-S.; Welbl, J.; Gowal, S.; Cherepanov, A.; Molloy, J.; Mankowitz, D. J.; Robson, E. S.; Kohli, P.; de Freitas, N.; Kavukcuoglu, K.; and Vinyals, O. 2022. Competition-level code generation with AlphaCode. *Science*, 378(6624): 1092–1097.

Lissandrini, M.; Mottin, D.; Palpanas, T.; and Velegrakis, Y. 2018. Data Exploration Using Example-Based Methods. *Synthesis Lectures on Data Management*, 10(4): 1–164.

Liu, B.; Jiang, Y.; Zhang, X.; Liu, Q.; Zhang, S.; Biswas, J.; and Stone, P. 2023. LLM+P: Empowering Large Language Models with Optimal Planning Proficiency. *CoRR*, abs/2304.11477.

Long, J. 2023. Large Language Model Guided Tree-of-Thought. *CoRR*, abs/2305.08291.

Menon, A.; Tamuz, O.; Gulwani, S.; Lampson, B.; and Kalai, A. 2013. A Machine Learning Framework for Programming by Example. In Dasgupta, S.; and McAllester, D., eds., *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, 187–195. Atlanta, Georgia, USA: PMLR.

Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; and Xiong, C. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. arXiv:2203.13474.

Pearl, J. 1984. *Heuristics: intelligent search strategies for computer problem solving*. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN 0201055945.

Psallidas, F.; Ding, B.; Chakrabarti, K.; and Chaudhuri, S. 2015. S4: Top-k Spreadsheet-Style Search for Query Discovery. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2001–2016. New York, NY, USA: Association for Computing Machinery. ISBN 9781450327589.

Qi, J.; Tang, J.; He, Z.; Wan, X.; Cheng, Y.; Zhou, C.; Wang, X.; Zhang, Q.; and Lin, Z. 2022. RASAT: Integrating Relational Structures into Pretrained Seq2Seq Model for Text-to-SQL.

Saparov, A.; and He, H. 2023. Language Models Are Greedy Reasoners: A Systematic Formal Analysis of Chain-of-Thought. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Schlag, I.; Sukhbaatar, S.; Celikyilmaz, A.; Yih, W.; Weston, J.; Schmidhuber, J.; and Li, X. 2023. Large Language Model Programs. *CoRR*, abs/2305.05364.

Scholak, T.; Schucher, N.; and Bahdanau, D. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 9895–9901. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics.

Seneff, S.; Glass, J.; Goddeau, D.; Goodine, D.; Hirschman, L.; Leung, H.; Phillips, M.; Polifroni, J.; and Zue, V. 1991. Development and Preliminary Evaluation of the MIT ATIS System. In *Proceedings of the Workshop on Speech and Natural Language*, HLT '91, 88–93. USA: Association for Computational Linguistics.

Shen, Y.; Chakrabarti, K.; Chaudhuri, S.; Ding, B.; and Novik, L. 2014. Discovering Queries Based on Example Tuples. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, 493–504. New York, NY, USA: Association for Computing Machinery. ISBN 9781450323765.

Shen, Y.; Song, K.; Tan, X.; Li, D.; Lu, W.; and Zhuang, Y. 2023. HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face. In Oh, A.; Neumann, T.; Globerson, A.; Saenko, K.; Hardt, M.; and Levine, S., eds., *Advances in Neural Information Processing Systems*, volume 36, 38154–38180. Curran Associates, Inc.

Tabassum, A.; and Patil, R. R. 2020. A survey on text pre-processing & feature extraction techniques in natural language processing. *International Research Journal of Engineering and Technology, June*.

Tran, Q. T.; Chan, C.-Y.; and Parthasarathy, S. 2014. Query Reverse Engineering. *The VLDB Journal*, 23(5): 721–746.

Wang, B.; Shin, R.; Liu, X.; Polozov, O.; and Richardson, M. 2019. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. *CoRR*, abs/1911.04942.

Wang, C.; Cheung, A.; and Bodik, R. 2017. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. *SIGPLAN Not.*, 52(6): 452–466.

Wang, J.; Ng, P.; Li, A. H.; Jiang, J.; Wang, Z.; Nallapati, R.; Xiang, B.; and Sengupta, S. 2022. Improving Text-to-SQL Semantic Parsing with Fine-grained Query Understanding. arXiv:2209.14415.

Wang, Y.; Wang, W.; Joty, S. R.; and Hoi, S. C. H. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *CoRR*, abs/2109.00859.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Ichter, B.; Xia, F.; Chi, E. H.; Le, Q. V.; and Zhou, D. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In Koyejo, S.; Mohamed, S.; Agarwal, A.; Belgrave, D.; Cho, K.; and Oh, A., eds., *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Wolf, T.; Debut, L.; Sanh, V.; Chaumond, J.; Delangue, C.; Moi, A.; Cistac, P.; Rault, T.; Louf, R.; Funtowicz, M.; Davison, J.; Shleifer, S.; von Platen, P.; Ma, C.; Jernite, Y.; Plu, J.; Xu, C.; Le Scao, T.; Gugger, S.; Drame, M.; Lhoest, Q.; and Rush, A. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 38–45. Online: Association for Computational Linguistics.

Yao, S.; Yu, D.; Zhao, J.; Shafran, I.; Griffiths, T.; Cao, Y.; and Narasimhan, K. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In Oh, A.; Naumann, T.; Globerson, A.; Saenko, K.; Hardt, M.; and Levine, S., eds., *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Yu, T.; Zhang, R.; Yang, K.; Yasunaga, M.; Wang, D.; Li, Z.; Ma, J.; Li, I.; Yao, Q.; Roman, S.; Zhang, Z.; and Radev, D. R. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. *CoRR*, abs/1809.08887.

Zhong, R.; Yu, T.; and Klein, D. 2020. Semantic Evaluation for Text-to-SQL with Distilled Test Suites. arXiv:2010.02840.