CatSQL: Towards Real World Natural Language to SQL Applications

Han Fu
Alibaba Group
fuhan.fh@alibaba-inc.com

Chang Liu
Alibaba Group
liuchang2005acm@gmail.com

Bin Wu
Alibaba Group
binwu.wb@alibaba-inc.com

Feifei Li
Alibaba Group
lifeifei@alibaba-inc.com

Jian Tan
Alibaba Group
j.tan@alibaba-inc.com

Jianling Sun
Zhejiang University
sunjl@zju.edu.cn

ABSTRACT

Natural language to SQL (NL2SQL) techniques provide a convenient interface to access databases, especially for non-expert users, to conduct various data analytics. Existing methods often employ either a rule-base approach or a deep learning based solution. The former is hard to generalize across different domains. Though the latter generalizes well, it often results in queries with syntactic or semantic errors, thus may be even not executable. In this work, we bridge the gap between the two and design a new framework to significantly improve both accuracy and runtime. In particular, we develop a novel *CatSQL* sketch, which constructs a template with slots that initially serve as placeholders, and tightly integrates with a deep learning model to fill in these slots with meaningful contents based on the database schema. Compared with the widely used sequence-to-sequence-based approaches, our sketch-based method does not need to generate keywords which are boilerplates in the template, and can achieve better accuracy and run much faster. Compared with the existing sketch-based approaches, our *CatSQL* sketch is more general and versatile, and can leverage the values already filled in on certain slots to derive the rest ones for improved performance. In addition, we propose the *Semantics Correction* technique, which is the first that leverages database domain knowledge in a deep learning based NL2SQL solution. *Semantics Correction* is a post-processing routine, which checks the initially generated SQL queries by applying rules to identify and correct semantic errors. This technique significantly improves the NL2SQL accuracy. We conduct extensive evaluations on both single-domain and cross-domain benchmarks and demonstrate that our approach significantly outperforms the previous ones in terms of both accuracy and throughput. In particular, on the state-of-the-art NL2SQL benchmark Spider, our *CatSQL* prototype outperforms the best of the previous solutions by 4 points on accuracy, while still achieving a throughput up to 63 times higher.

PVLDB Reference Format:
Han Fu, Chang Liu, Bin Wu, Feifei Li, Jian Tan, and Jianling Sun. *CatSQL*: Towards Real World Natural Language to SQL Applications. PVLDB, 16(6): 1534 - 1547, 2023.
doi:10.14778/3583140.3583165

Proceedings of the VLDB Endowment, Vol. 16, No. 6 ISSN 2150-8097.
doi:10.14778/3583140.3583165

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/asfuhan/CatSQL.git.

1 INTRODUCTION

In the era of big data, the wide adoption of database management systems (DBMS) has grown from traditional companies such as banks to almost all industries. The DBMS users also extends from well-trained database experts to data analysts who may not be familiar with the underlying database schemas, or even non-expert users who do not have the understanding of the basic DBMS concepts. A traditional DBMS provides SQL as a convenient interface to access the data stored in the DBMS. However, writing a correct SQL may be challenging for a non-expert user, especially for complex tasks.

It is a long standing open question whether it is possible to build a natural language to SQL (NL2SQL) system so that users can express their query intentions with natural language, such as English, and rely on a NL2SQL system to successfully translate to correct SQL statements. Researchers from the database community have utilized rule-based methods to tackle this problem [2, 26, 36, 38]. Existing systems first parse the natural language question into an intermediate representation (e.g. a parsing tree), and then develop rules to map it into a SQL abstract syntax tree, which is then converted into a SQL query. Such a method works well on a targeted domain, but is hard to generalize across different domains and tasks. Therefore, deploying such a system requires tremendous human efforts to develop new rules to adapt to a new domain.

Recent advancements from the deep learning (DL) community and the natural language processing (NLP) community demonstrate promising results towards tackling the cross-domain adaption problem. A deep learning model is a machine learning model with a large set of trainable parameters. Given a large dataset, the parameters can be trained to generalize well to unseen examples. The NLP community creates several large NL2SQL datasets, e.g., WikiSQL [58] and Spider [55]. Using these datasets, researchers have proposed deep learning-base NL2SQL solutions that can achieve a relatively high accuracy, i.e., 80% to 90%, on these benchmark datasets [8, 13, 19, 29, 35, 37, 43, 49]. When they are handling relatively complex queries, however, their performance drops significantly, down to below 50%. The reason why deep learning approaches do not show good performance on complex queries is because they treat the NL2SQL problem as a machine translation

problem and solely rely on machine translation techniques, such as sequence-to-sequence, to solve it. These techniques do not leverage database domain knowledge, and are easy to introduce semantic errors when the question needs a good understanding of the underlying database schemas.

In this work, we are the first to bridge the advancements from both database and deep-learning communities, develop novel approaches to the NL2SQL problem, and demonstrate such a combination can boost the performance significantly. In particular, our approach, named *CatSQL*, is a deep learning-based approach, which can mitigate the issues of rule-based solutions to generalize across application domains. Our *CatSQL* approach differs from existing deep learning approaches in two different aspects. First, most existing deep learning approaches are based on a sequence-to-sequence [37, 49, 58] or sequence-to-tree model [8, 13, 16, 35, 43]. These methods do not guarantee the generated SQL queries are executable or even syntactically legal. Instead, our approach is a sketch-based solution, which relies on our novel *CatSQL* sketch to generate the SQL query. *CatSQL* sketch is a template with keywords and slots. These slots initially serve as placeholders. We use a deep learning model to fill in the empty slots to get a final SQL query, which is almost always a legal SQL query.

In doing so, the deep learning model in our solution can focus on generating essential information relevant to the natural language question to fill in the slots, instead of wasting resources to generate boilerplates such as keywords. Notice that there also exist works that use such a sketch-based idea [9, 20, 48, 54]. These solutions typically develop different specialized templates for different application domains [9, 20, 48, 54]. These templates are not general enough, and limit the expressiveness of SQL queries that can be handled. Our *CatSQL* sketch is more general and largely akin to the standard SQL syntax. Also, existing works typically fill in different slots independently and train separate sets of parameters for different query modules. Our *CatSQL* SQL generation algorithm employs a novel *CatSQL* sketch, which is general enough, and can facilitate the idea of parameter sharing to boost the performance.

Second, although some deep learning approaches are designed to generate syntax legal SQL statements, the predicted queries are not guaranteed to be semantically legal to return non-empty results. This is because none of existing deep learning-based solutions take the schema information of the underlying database into consideration. Considering semantic constraints in a neural network is inherently challenging, since the former handles discrete rules, while the latter solves continuous optimization problems. To mitigate this issue, we develop a novel *Semantics Correction* algorithm to examine whether the generated SQL queries are semantically unsound and try to fix the semantic issues in the generated queries. In doing so, *Semantics Correction* leverages database domain knowledge into the SQL generation process to avoid obvious semantic errors, so as to greatly improve the performance for generating complex queries. To the best of our knowledge, we are the first to incorporate semantic information into a deep learning-based NL2SQL solution. We evaluate our *CatSQL* approach on the well-known Spider dataset, and the results demonstrate that *CatSQL* is 4 points better than the previous state-of-the-art methods.

We summarize our contributions as follows:

- A novel sketch-based model *CatSQL* is proposed to achieve the state-of-the-art performance on various NL2SQL benchmarks;
- *Semantics Correction* of *CatSQL* is the first work that adopts database domain knowledge in a NL2SQL solution;
- Extensive evaluations demonstrate that *CatSQL* significantly outperforms all existing solutions on cross-domain benchmarks such as Spider and WikiSQL;
- On single-domain benchmarks, *CatSQL* solution also significantly outperforms existing solutions;
- *CatSQL* prototype achieves outstanding runtime performance: its single query runtime latency can be $2\times - 20\times$ faster than all baselines; while its throughput can be $2.5\times - 63\times$ higher than the previous approaches.

The rest of the paper is organized as follows. In Section 2, we first demonstrate the problem with a motivating example, provide necessary backgrounds and terminologies for existing NL2SQL techniques, and give an overview of our *CatSQL* solution. Next, we present our *CatSQL* SQL generation and the *Semantics Correction* approach in Section 3. We explain our system design choice and model details in Section 4, and evaluate our system in Section 5. We discuss related works in Section 6, and conclude this paper in Section 7.

2 OVERVIEW

In this section, we first introduce the NL2SQL problem using an example. Then we introduce some background knowledge so as to explain the existing NL2SQL solutions; we also explain the issues of existing methods and why gap still exists when applying them for real-world applications. In the end, we describe our overall design of *CatSQL*.

2.1 Motivating Example

Figure 1 presents an example to translate a natural language question into its corresponding SQL query. The query is looking for average life expectancy of some countries which does not use English as its official language. Figure 2 illustrates a part of the database schema relevant to the question. The database contains several tables: “Country”, “Country Language”, “City”, etc. Each table has several columns, such as “Name" in “Country”, “IsOfficial” in “CountryLanguage”, etc. We notice that both table names and column names are meaningful phrases. This makes the NL2SQL task possible. Such a naming practice is also well adopted in modern database applications.

The NL2SQL problem is to translate the natural language question into the corresponding SQL query. We need to figure out three challenging questions from the natural language description: (1) what tables and columns that will be used in the query; (2) what is the correct query structure; and (3) how to fill in query details and the literal in the query. In the example in Figure 1, from the description, we can identify two tables whose names are relevant: “Country” and “CountryLanguage”. However, the second table will be used in the nested query, so the outter query will use only the “Country” table.

The query structure is a nested query structure: the outter query calculates the average life expectancy, which corresponds to the

What is average life expectancy in the countries where English is not the official language?

```
SELECT avg(LifeExpectancy)
FROM Country
WHERE Name NOT IN (
SELECT T1.Name FROM Country AS T1
JOIN CountryLanguage AS T2
ON T1.Code = T2.CountryCode
WHERE T2.Language = "English"
AND T2.IsOfficial = True
)
```

Figure 1: A natural language question and its corresponding SQL query.

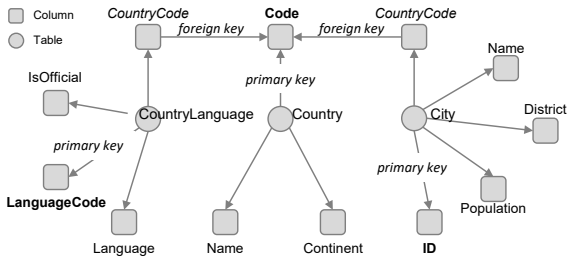

Figure 2: The schema graph of the database that the query in Figure 1 is executed on. Each circle is a table, and each box is a column. An arrow from a circle to a box indicates that a table contains a column; an arrow labeled with "primary key" from a circle to a box is a primary key relationship; and an arrow labeled with "foreign key" from one box to another box is a foreign key relationship.

main clause; the nested query quantifies the selected countries use English as their official language, which corresponds to the attribute clause. In the end, we need to translate "average life expectancy" into `avg(LifeExpectancy)`; and also, we need to fill in `"English"` into the `T2.Language = "English"` constraint, and `True` into the `T2.IsOfficial = True` constraint. This requires both understanding the question semantics and understanding the data stored in the database.

Solutions from the DB community requires designing rules for the mapping between natural language tokens into SQL elements, and translating a natural language parsing tree into a SQL's abstract syntax tree (AST). However, these approaches' performance decay significantly when they are applied to a new domain of database, that requires redesigning the mapping.

The state-of-the-art solutions from the NLP community relies on deep neural network to train a large model, which can be applied to handle the cross-domain issue, and demonstrate superior performance. However, all of these approaches do not leverage any semantics information to generate the query. Therefore, when these approaches are applied to handle complex query structures, their performances decay significantly. Also, since large neural networks require computing expensive computation, their runtime performance typicaly increases with the increase of the model's parameters.

Our work combines the benefits of the approaches from the two community, and we demonstrate that our approach performs better than all existing solutions. Our approach employs a deep learning architecture, and we leverage DB domain knowledge to develop novel *Semantics Correction* techniques to postprocess the query generated by a neural network to fix obvious semantics errors. In the following, we will first provide necessary background on how a deep neural network NL2SQL solution works, and then we will give an overview of our approach.

2.2 Background

Most of the state-of-the-art NL2SQL algorithms are built on top of deep neural networks. In this section, we will briefly introduce these building blocks with their key ideas to faciliate the readers to understand our approach. In particular, we will use the example in Figure 1 to illustrate how a deep neural network-based NL2SQL solution works.

Embedding. As an atomic operator, neural networks, such as LSTMs [18] and Transformers [41], can model a sequence of tokens into a sequence of N-dimensional numeric vectors called *embedding*. Formally, given an input sequence of tokens $t_1, ..., t_n$, a model $\mathcal{M}$ converts them into a sequence of embeddings $h_1, ..., h_n$, i.e.,

$$\mathcal{M}(t_1, ..., t_n) = h_1, ..., h_n$$

Note that each embedding h_i encodes information of position i, that includes not only the token t_i at this position, as well as all other tokens that may have information towards understanding t_i and the context input.

The network to compute embeddings is also referred to as the *encoding network*. The natural language community makes tremendous efforts on the general purpose pretrained encoding networks, and the latest results for NL2SQL are BERT[11] and GraPPa[53] which are widely adopted by the state-of-the-art NL2SQL models.

Classifier. Using this operator, we can build a classifier to classify the input sequence into one of C categories as follows. Let us say

$$f(t_1, ..., t_n) = \mathbf{softmax}\big(W \sum_{}^{n} \mathcal{M}(t_1, ..., t_n)\big)$$

where W is $C \times N$ matrix, and $\mathbf{softmax}(v)_i = v_i/(\sum_j v_j)$ is a well adopted deep learning operator. Therefore, $f(t_1, ..., t_n)$ will output a C-dimension vector, and the summation of all dimensions is 1. We can view $f(t_1, ..., t_n)$ as a probability distribution over the C categories. For this classification task, we can choose the dimension with the highest probability as the prediction. It is obvious that we can run another multi-layer neural network on top of $\mathcal{M}(t_1, ..., t_n)$ instead of a simple softmax-layer to perform the classification task. Actually, our work will run a four layer transformer instead.

Sequence-to-sequence translation. On top of this sequence classification example, we can build a *sequence generator* so that the overall model becomes a sequence-to-sequence generation algorithm [1, 40]. For example, given an input sequence $t_1, ..., t_n$, we want to translate it into an output sequence $d_1, ..., d_m$. Similar to

the classification problem, we essentially estimate the probability of $P(d_1, ..., d_m|t_1, ..., t_n)$, and choose the one with the maximal probability. However, this naive idea faces two obstacles: (1) the output sequence may be arbitrarily long, and therefore there can be infinite many possible outputs; and (2) even for a fix length of outputs, there are exponentially many candidates, and thus an exhaustive search approach is infeasible.

The deep learning community tackles this problem using a beam-search approach. The model first calculates the pool of candidates of d_1 using the decoding network method explained above. The beam search approach keeps only the top K possible candidates for d_1, where K is referred to as the beam size. When generating d_2, for each candidate d_1, we can estimate the probability of $P(d_2|d_1, t_1, ..., t_n)$ using the same approach, and calculate

$$P(d_1, d_2|t_1, ..., t_n) = P(d_2|d_1, t_1..., t_n) \cdot P(d_1|t_1..., t_n)$$

In doing so, we can obtain a candidate set of d_1, d_2 sequences, and we reserve only top K candidates with the highest probability in the candidate set. This routine can continue to generate arbitrarily long sequences. We use a special token $\langle$EOS$\rangle$ to mark the end of a sequence. When this token is generated at the end of a sequence, we no long extend it. The beam search procedure terminates when the candidate with the highest probability reaches the $\langle$EOS$\rangle$ token.

Subsequent works also extend this approach to generate the abstract syntax tree instead of a token sequence as the output[16, 43]. We do not elaborate the details, but refer the readers to [52] for more information.

Training. To train a neural network model, we can design a loss function $\mathcal{L}$ to estimate the distance between the prediction and the ground truth. Since deep neural network models are continuous functions, we can calculate the *gradient* $\partial\mathcal{L}/\partial\theta$, where θ is the set of all parameters used in the model. Then we can apply a gradient descent algorithm to update the parameters with $\theta \leftarrow \theta - \eta\frac{\partial\mathcal{L}}{\partial\theta}$, where η is a small value that is typically referred to as the *step size*. Different algorithms[7, 24, 28] will calculate η differently.

2.3 Existing NL2SQL Approaches

In this section, we explain three categories of NL2SQL solutions: (1) rule-based approaches; (2) sequence-to-sequence approaches; and (3) *sketch*-based approaches. We will briefly explain the basic ideas below.

2.3.1 Rule-based approaches. Existing NL2SQL solutions developed from the database community mainly relies on manually designed heuristics to translate a natural language question into a SQL query [2, 26, 36, 38]. In particular, these approaches use standard NLP techniques to parse the natural language question into a parsing tree; and then depending on the tree structure, rules are developed to translate the parsing tree into a SQL AST, which is then converted into the final SQL query. The subtle issue is how to generate the table/column names and string literals in the query, which can be found by looking into the database. To do so, these approaches develop a mapping between natural language token into elements in the queried databases to faciliate the translation procedure. The benefits of these approaches are: (1) they do not need training a machine learning model, and thus do not need to develop a training set; and (2) they can handle complex queries when the rules are designed so. However, when they are deployed to a new database domain, they require developing a novel mapping, which requires tremendous human efforts or the performance will decay significantly.

2.3.2 Sequence-to-sequence approaches. A sequence-to-sequence approach considers a SQL query as a sequence of tokens, so that any deep learning approach introduced in Section 2.2 can be applied. The main challenge is that the generated output sequence may not be a valid SQL statement. Therefore, almost all works along this line are designed to handle syntactically invalid SQL queries. For example, the state-of-the-art work along this line is called PICARD [37]. Its algorithm checks whether the generated partial SQL statement violates the syntax rules during the beam search and filters out invalid candidates. Another line of works consider to generate the abstract syntax tree (AST) using a sequence-to-tree approach. In doing so, any generated AST automatically becomes syntactically legal. For example, most existing works along this line are based on RAT-SQL [43]. These approaches typically face the problem of picking a wrong column or a wrong value. Therefore, most of the efforts in these works are devoted to tackling this issue.

2.3.3 Sketch-based deep learning approaches. Another alternative line of deep learning NL2SQL works develop *sketch*-based methods. Typically, a sketch is a SQL template with empty slots so that neural networks only need to fill in those slots. A sketch-based method is inherently faster than a sequence-to-sequence solution, because these models do not need to concern about predicting SQL keywords which are boilerplates in the sketch. Instead, a sketch-based model can focus on filling in the essential information that is extracted from the natural language question. However, existing works [48, 54] do not achieve the state-of-the-art performance along this line. In this work, we develop a novel sketch-based approach called *CatSQL*, and demonstrate that it can achieve a superior performance than all existing methods.

3 CATSQL APPROACH

In this section, we introduce our *Column Action Templates* approach to tackle the NL2SQL problem. Our approach is a template-based deep learning approach. That is, using a deep neural network to fill in the empty slots in the template to form the final SQL query. Compared with existing sequence-to-sequence-based approach, the template-based approach does not need to waste resources to generate keywords such as **SELECT**/**FROM**/**WHERE**; therefore, such an approach can focus on generating essential information to form a SQL query.

Existing template-based approaches suffer from two issues. First, they train different models to generate different sub-clauses. Therefore, each model uses only a subset of the data. For example, the model for the **SELECT** clause does not leverage any data for the **WHERE** clause. Existing work demonstrates that, for a sequence-to-sequence model, sharing model parameters to generate different clauses will improve the performance.

In our work, we develop a special template called *Column Action Templates SQL* or *CatSQL*, so that we only train one model for multiple clauses to implement the idea of parameter sharing for a template-based approach. Second, existing works rely on pure deep

aquery	=	**SELECT** [CAT]+ **FROM** [table | nested_term_token]+ [**WHERE** | **HAVING**] [CAT]* **GROUP BY** [CAT]* **ORDER BY** [CAT]* [**LIMIT** literal]?
CAT	=	AGG? **DISTINCT**? column OP? value? [**AND** | **OR**]? [**ASC** | **DESC**]?
value	=	literal | (literal, literal) | nested_term_token
query	=	aquery | aquery conj query
conj	=	**INTERSECT** | **UNION** | **EXCEPT**

Table 1: *CatSQL* template

neural network to generate the query, and thus ignore semantics constraints such that most join conditions should have a PK-FK (primary key-foreign key) relationship. Our *CatSQL* approach develops a novel *Semantics Correction* technique to take care of these semantics constraints. Our *Semantics Correction* technique fixes simple semantics errors during the *CatSQL* SQL generation process, and post-processes the generated query to fix more complex semantics errors considering the database schema.

In the following, we will first introduce our *CatSQL* template (Section 3.1), and then describe our neural network model to fill in each slots in the template (Section 3.2). In the end, we explain our *Semantics Correction* technique in Section 3.3.

3.1 CatSQL template

The *CatSQL* template is defined in Table 1. The core definition of *Column Action Templates* (or CATs) is as follows:

AGG? **DISTINCT**? *column* OP? value? [**AND** | **OR**]? [**ASC** | **DESC**]?

We can see that CAT is a generic template that fits into multiple causes of a SQL query including: (1) **SELECT**; (2) **WHERE**; (3) **GROUP BY**; (4) **ORDER BY**; and (5) **HAVING**. Different clauses fill only a subset of slots. For example, the **SELECT** fills (optionally) AGG, **DISTINCT**, and *column*; and **ORDER BY** fills *column*, and [**ASC**|**DESC**].

Each of these clauses is a keyword followed by a sequence of CATs. In particular, for the **WHERE** clause, the conjunction keywords (i.e., **AND** and **OR**) are treated as a part of a CAT. For example, the following expression

$$\texttt{a} > 1 \textbf{ AND } \texttt{b} < 10 \textbf{ OR } \texttt{c} = 20$$

is composed of the following three CATs:

$$(1)\texttt{a} > 1 \textbf{ AND} \qquad (2)\texttt{b} < 10 \textbf{ OR} \qquad (3)\texttt{c} = 20$$

The definition of the *CatSQL* template is designed to facilitate the idea of parameter sharing, which is not adapted by previous sketch-based approaches. In particular, since each of the four CAT clauses can be viewed as a sequence of CATs, we can train one sequence-to-sequence model for all four different clauses. At runtime, once the CATs are predicted, we can simply construct the final SQL query by assembling different CAT clauses. We will explain how our neural network fills a *CatSQL* sketch in the next section.

3.2 *CatSQL* query generation

As illustrated in Figure 3, the overall architecture of *CatSQL* is composed of four components, namely (1) GraPPa embedding network, (2) CAT decoder network, (3) conjunction network, and (4) FROM decoder network. We now explain each of these components.

GraPPa Embedding Network. The first implementation of the parameter sharing idea is through the use of a common embedding network. To this aim, we concatenate the natural language question, database schema, and some additional information used for generating nested queries. The question is a sequence of tokens embraced between a pair of a [CLS] token and a [SEP] token. The database schema part encodes each table's name and its column names together into a table schema sequence, and concatenate all tables' schema sequences together separated by a [SEP] token. The additional information is mainly used for generating nested queries, so we will leave them for shortly later.

We run an embedding network called GraPPa [53] over the input sequence to generate a sequence of hidden states. The GraPPa architecture is a 24-layer transformer network pre-trained with multiple SQL parsing tasks. All other components share the same GraPPa encoding network so as to leverage the parameter sharing idea.

Generating the query body using the CAT Decoder Network. For the five CAT clauses, we can model each of them as a sequence of CATs with some slots masked out. Therefore, predicting each clause is formulated as a standard CAT sequence generation problem. In particular, from Table 1, a query can be viewed as the concatenation of the **SELECT**, **FROM**, **WHERE**, **GROUP BY**, and **ORDER BY** clauses. In our framework, we handle the **FROM** clause differently, and each other clauses can be viewed as a sequence of CATs. In our approach, we concatenate these four clauses together separated by a special [SEP] token, thus we can view the problem of generating these four clauses as a standard sequence-to-sequence generation problem where the input sequence is described above, and the output is a sequence of CATs and [SEP] tokens.

To this aim, from the embedding sequence generated by the GraPPa Encoding Network, the CAT Decoder Network uses another four layer transformer network to generate an output sequence of CAT hidden states. Each of them corresponds to one CAT. Based on the generated CAT and [SEP] sequence, we can split each CAT into different clauses.

Each CAT has up to seven slots, and thus we use seven different classifiers to generate the differents lots of a CAT. Different clauses may fill different slots. For example, for the **SELECT** clause, we only fill the AGG, **DISTINCT**, and *column* slots, and ignore the outputs of the other four classifiers to make sure the generated SQL query is valid.

For 5 out of the 7 slots such as AGG, **DISTINCT**, OP, **AND**/**OR** conjunction, and **ASC**/**DESC** descriptor, we use a feed-forward layer with softmax activation as the classifier. If one slot is optional, we may fill a special [EMPTY] token to indicate that the slot is omitted.

The column slot is filled using a pointer newtork [42]. That is, for each CAT we first get the hidden state vector calculated by the CAT Decoder Network. At the same time, we collect all

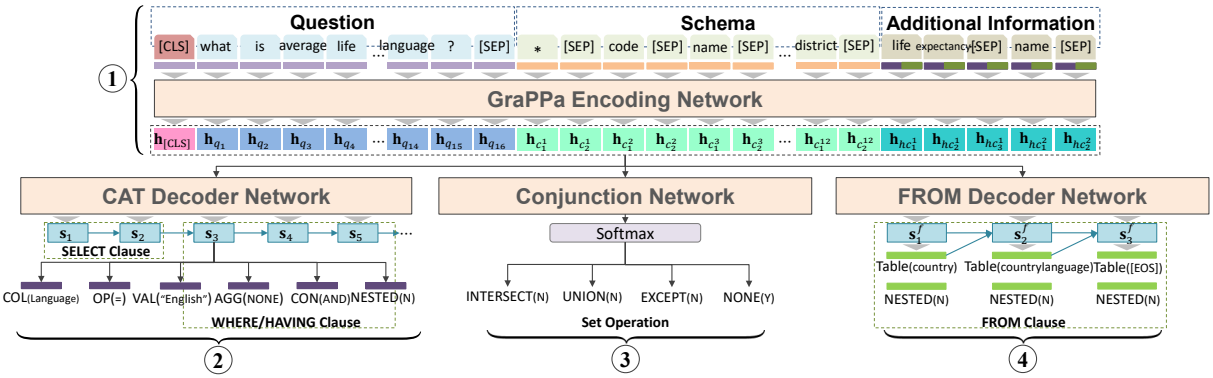

Figure 3: Architecture of *CatSQL*

columns in the database, and compute one embedding vector for each column name. Then for each column, we compute the score as the inner product of the CAT hidden state vector and the column's embedding vector. Then for each column name, we compute a score as the inner product of the CAT hidden state vector and the column embedding. The column name with the highest score is picked to fill in the `column` slot.

Before discussing the `value` slot handling approach, we want to mention that different clauses may have different slots voided. For example, when we fill in the **SELECT** clause, we do not need to fill in the slots such as `OP` and `value`. In this case, the network still generate the values for these slots, but will mask them out when forming the final query. We have a special routine to handle the . We now describe our approach below.

Literal handling. The `value` slot may have three different forms: (1) one single `literal`; (2) a pair of `literals`; and (3) the `nested_term_token`. When filling in this slot, the model will first run a classifer to determine which one of the three scenarios it chooses. When a `literal` is needed, a special literal handling routine will be employed.

Most existing literal filling works try to locate a token directly in the natural language question and copies it into the `literal` location. However, this approach suffers from the issue that the words used in the question may not match the values in the database when the exact data records are inaccessible to end users. Our approach will examine database values to improve the accuracy. For each literal slot to be generated, the network produces a embedding vector for this slot as the *literal embedding*. We also obtain the column in the same `CAT`. Then we retrieve all distinct values from this column, and calculate an embedding for each value. We calculate a cosine similarity score between the literal embedding and each value embedding vector, and the value with the maximal score is picked. If this score exceeds a manually specified threshold, then we copy this value directly; otherwise, we resort to the traditional approach to copy a token in the natural language question that best matches this literal embedding. We observe that this approach significantly improves the execution accuracy.

Note that the set of all distinct values in each column and their embeddings are computed offline, and are periodically updated with a model update. Therefore, the online serving module only needs to scan through this set of value embeddings. This takes much shorter time compared with the entire neural network computation.

Nested queries and query conjunction. We now explain how we expand a `nested_term_token` and how the Conjunction Network handles conjunction queries. When a `nested_term_token` is generated, the generator should expand it into a nested subquery. The approach is quite similar to the top-level process, but we provide additional information as an input to the GraPPa Network.

One straightforward idea is to use the entire top-level query (or outer block for the inner block in a nested query) to form a sequence as the additional information. However, this approach makes the additional input too long, and thus requires a large amount of computational resources, which limits the application in real-world scenes. In our framework, we propose an efficient method to record the information of the outer block. In particular, when expanding a `nested_term_token`, we collect all column names appeared in the outer query, and concatenate them into a sequence separated by the `[SEP]` tokens to form the additional information.

We handle the conjunction keywords such as **INTERSECT**, **UNION**, **EXCEPT**, and **LIMIT**, similar to expand the nested term token. Once an atomic query is generated, we run a Conjunction Network (part 3 in Figure 3) to predict whether a keyword exists for each of these four keywords. The model consists of a feed forward layer followed by an attention pooling layer and another feed forward layer. If **LIMIT** exists, a separate literal handling model generates the literal for this **LIMIT** clause. When any of the three conjunction keywords exist, e.g., **INTERSECT**, we consider the right part of the keyword as a nested query, and the left part as its outter query. In doing so, we delegate the generation of the right part to the above explained nested query generation routine.

FROM clause. The **FROM** clause is handled differently from other `CAT` clauses. It needs to generate two parts: (1) the tables to appear in the query; and (2) the join-conditions. Our FROM Decoder Network focuses on generating the tables in the **FROM** clause. In particular, we consider the tables in the **FROM** clause as a sequence of tables, and thus generating the clause is a standard sequence generation problem.

Our FROM Decoder Network employs a standard four layer transformer architecture to generate a sequence of table embeddings. From each embedding, we will first run a standard two-class classifier to classify whether it should be a table or a `nested_term_token`. If later, we will employ the nested query expansion technique discribed above to handle it.

If an embedding is a table, we employ a pointer network to select one table from all tables in the database. This is quite similar to the one used in the CAT Decoder Network to generate the `column` slot. The only difference is that the network here selects one table, instead of one column.

Our join condition generation algorithm does not use any learning algorithm, but runs a minimal spanning tree algorithm over the generated tables for the **FROM** clause. Once we generate a sequence of k tables, i.e, t_1, t_2, ..., t_k, we want to generate the join conditions to connect them. To this aim, we build a graph with k nodes, so that each table t_i corresponds to one node n_i in the graph. For two tables t_i and t_j, if we have a PK-FK relationship between t_i and t_j, we build an edge between n_i and n_j, labeled with a weight $|i-j|$ indicating the distance between the two tables in the **FROM** clause. Also, if t_i and t_j are the same table, we build an edge connecting n_i and n_j with the weight $|i-j|$. In this graph, each edge indicates either a PK-FK join condition or a self-join condition.

On this graph, we run a minimal spanning tree algorithm to get a set of $k-1$ edges with the minimal total weight. If the graph is not connected, then we get one minimal spanning trees for each connected subgraph. All edges on the minimal spanning tree will be converted into a join-condition in the generated query.

Some existing methods [9, 43] also employ the similar approach when generating the table sequence, but differs in how join condition is generated. For each generated table t_i, existing models try to find the PK-FK relationship between t_{i-1} and t_i and use it as the join condition. When this routine fails, it does not generate the join condition for t_i. Note that since the weight between two tables t_i and t_j is $|i-j|$, if t_i has a PK-FK relationship with t_{i-1}, then it will prefer choosing this relationship (which has the minimal weight of 1). Therefore, when existing methods generate the correct **FROM** clause, our algorithm behaves the same. At the same time, our approach generalizes better when such a relationship does not exist. In case, we need to seek a farther PK-FK relationship or a self-join relationship to form the join condition.

Once the **FROM** clause is predicted, a atomic SQL query is constructed by assembling all different `CAT` clauses and the **FROM** clause. If the `nested_term_token` presents in the query, the model is called iteratively until all nested tokens are expanded and no new nested token is predicted.

3.3 Semantics Correction

In this section, we present our *Semantics Correction* technique, which significantly improves the accuracy by leveraging database domain knowledge. While a deep learning model is effective in understanding the intention of a question, sometimes the generated SQL query expresses the same intention, but is semantically invalid considering the database schema.

It is challenging to incorporate these semantics constraints into a neural network-based SQL generation process. The reason is that a neural network approach is generally a continuous optimization approach, but semantics constraints are mostly discrete rules to satisfy. Our *Semantics Correction* technique is developed as a rule-based approach to postprocess the generated query, and makes its best-effort attempt to fix obvious semantics errors. We classify our *Semantics Correction* rules into three categories: (1) token-level violation, (2) **FROM** clause revision, and (3) join-path revision.

We want to emphasize that our *Semantics Correction* is a best-effort attempt, and does not guarantee to generate a correct query. Since it only affects queries that involve semantics error, it will not rewrite a correct query into a wrong one. Therefore, our *Semantics Correction* always improve the accuracy. We detail the three categories of rules below.

Token-level violation. This type of semantics errors are the easiest to fix. Let us see the following example:

```
SELECT AVG(petType) FROM pets GROUP BY petType
```

Since the column `petType` takes categorical values, it does not make sense to compute the average over this column. In this case, we consider the `AGG` slot in the `CAT` corresponding to `AVG(petType)` cannot take the value of `AVG`. Our token-level violation rules will mask such an invalid value with an output probability 0 during the decoding process.

This type of violation is easy to handle, since it only requires inspecting a small part of the partially generated query, and does not need a deep understanding of the entire query.

FROM clause revision. This type of violation is that the query body uses some columns that do not appear in any tables in the **FROM** clause. In this case, we prefer to consider the **WHERE** clause is more trustworthy since it captures more semantics of the natural language question. Therefore, we will add the missing tables to the end of the **FROM** clause. Also, we will generate the join-conditions for the newly added tables. We can employ the same minimal spanning tree technique to generate these join conditions.

Join path revision. In many cases involving nested query, it is very easy that the join path is wrong. Let us see the following SQL query:

```
SELECT avg(LifeExpectancy) FROM country
WHERE name NOT IN (SELECT isofficial
FROM countrylanguage
WHERE language = "english" )
```

In this example, the column `name` in the `country` table does not have a FK relationship with the `isofficial` column in the `countrylanguage` table generated in the nested query. In this case, we will try to reconstruct a correct join path.

In particular, when a query involves a component of `c1 op (` **SELECT** `c2` **FROM** `t2 )`, we require `c1` and `c2` have a PK-FK relationship. When this requirement is not met, we try to rewrite it with minimal modifications.

In particular, in such a case, we assume (1) `c1` in the outter query is correct; and (2) the column `c2` and `t2` in the nested query are correct. Based on these two assumptions, we need to reconstruct a join path connecting `c1` to `t2` from the schema graph constructed similarly as described for constructing the join conditions for the **FROM** clause.

In doing so, the above example will be rewritten into the following:

```
... name NOT IN (SELECT country.name
FROM countrylanguage JOIN country ON
countrylanguage.countrycode = country.code ...)
```

4 SYSTEM IMPLEMENTATION

In this section, we present details on system implementation. Our prototype uses MySQL v5.7.30 to host our databases and all generated SQL queries will be executed by the MySQL engine.

Offline processing. The offline procedure includes model training and some pre-processing procedures. We perform a pre-processing procedure for literal handling. We use Redis [34] as our in-memory key-value store, and we store a mapping from literals to their embeddings in one redis instance. In particular, for each column in the table, we retrieve all values, and use word2vec [31] to compute an embedding for each of them. Here, we choose to use word2vec instead of other more recent language models such as BERT mostly for speed. We observe that word2vec runs much faster than BERT, while achieving comparable performance. We then use each column-value pair as the key and its embedding as the value to store them into another redis store.

Online serving. Our online serving module has a web application built on top of RESTful APIs to accept natural language questions. The backend launches a separate process for each query to perform SQL generation and *Semantics Correction* in a sequel. The bottleneck of parallelization is at the GPU memory. Each process use the minimum amount of GPU memory so as to maximize the number of queries to be handled in parallel. In each process, the SQL generation module retrieves all value-embedding pairs into the memory for literal handling. For all datasets in our evaluation, the time for literal handling takes less than 1 millisecond, which is negligible compared with time taken for the entire process. Other modules are implemented as we described in Section 3.

Model details. Both CAT Decoder Network and FROM Decoder Network consist of four transformer decoder layers. All hidden states are 256-dimensional vectors. We use Adam [24] to train our model with default setup. We use different learning rates for different components. For *CatSQL*, the encoder's learning rate is set to be 7×10^{-6}, and the transformer decoders' is set to 3×10^{-4}.

5 EXPERIMENTS

In this section, we evaluate the performance of our system in terms of both accuracy and running speed compared with previous state-of-the-art methods. In the following, we will first explain the evaluation setup. We then present the accuracy results with respect to different modules of the system. This is followed by the running speed and throughput evaluation results. In the end, we study some cases which shed light on the direction of future works.

5.1 Benchmarks and Baselines

Dataset. In this work, we adopt two recent cross-domain NL2SQL benchmarks, WikiSQL [58] and Spider [55], which is widely evaluated in the NLP community. WikiSQL is the largest public NL2SQL dataset containing 80,000 human-annotated queries on 24,000 tables over multiple domains. The drawback of WikiSQL is that its queries only use **SELECT** and **WHERE**, while the **FROM** clause is provided and no other complex operators are involved.

Spider is considered as the hardest NL2SQL dataset currently. Spider supports much richer SQL syntax, and it classifies the dataset into four categories based on their hardness levels:

- Easy. Single-table queries;
- Medium. Using **GROUP BY**, **ORDER BY** and **HAVING**;
- Hard. Using **JOIN** on multiple tables and set operations;
- Extra Hard. Nested queries.

Both WikiSQL and Spider splits the data into `train/dev/test`. `train` is used for training the model; `dev` is used to evaluate the model's performance during training to avoid overfitting; and `test` is used for evaluation after training. Spider has a separate hold-out test dataset that is not publicly available. To avoid ambiguity, in this work, all reported results are evaluated on the publicly available dataset only.

Besides, we also use several classical single-domain NL2SQL benchmarks, including GeoQuery [21, 32, 56, 57], Scholar [21], MAS [26], IMDB [50], and YELP [50] to make a fair comparison of our *CatSQL* approach and existing NL2SQL systems. The statistics are listed in Table 2

Evaluation metrics. To evaluate the accuracy of our approach, we mostly focus on the so-called *execution accuracy* (i.e., Acc_{ex}) metrics. For each test case, we will run different systems to get a

Table 2: Statistics for NL2SQL benchmarks.

Benchmark	Queries	Tables	Databases	Rows
GeoQuery [56]	880	7	1	937
Scholar [21]	816	10	1	144M
MAS [26]	196	17	1	54.3M
IMDB [50]	131	16	1	39.7M
YELP [50]	128	7	1	4.48M
WikiSQL [58]	80,654	26,531	26,531	459K
Spider [55]	9,693	873	200	1.57M

Table 3: Baselines to compare

Method	Reason to include	Pretraining
PICARD [37]	The top-1 on Spider Acc_{ex}	T5-3B [33]
NatSQL [13]	The top-2 on Spider Acc_{ex}	GraPPa [53]
SmBoP [35]	The top-3 on Spider Acc_{ex}	GraPPa [53]
LGESQL [8]	The top-2 on Spider Acc_{lf}	ELECTRA [10]
RAT-SQL [43]	Widely adopted baseline	BERT [11]
RYANSQL [9]	The SOTA sketch-based model	BERT [11]
SeaD [49]	Top-1 on WikiSQL board	BART [25]
SDSQL [19]	Top-2 on WikiSQL board	BERT [11]
HydraNet [29]	Top-4 on WikiSQL board	RoBERTa [27]
SQLova [20]	We can obtain code and model	BERT [11]

Table 4: Evaluation on Spider with different hardness levels. *CatSQL* (w.o. SC) does not perform *Semantics Correction*. *CatSQL* (w.o PC) does not use parameter sharing.

Method	Logical From Accuracy					Execution Accuracy					Executable Rate
	Easy	Medium	Hard	Extra Hard	All	Easy	Medium	Hard	Extra Hard	All	
NaLIR	0.8	0.2	0.0	0.0	0.3	1.6	0.7	0.0	0.0	0.7	3.9
Templar	0.8	0.2	0.0	0.0	0.3	1.6	0.4	0.6	0.0	0.7	3.7
RYANSQL	87.9	68.2	54.0	42.8	66.4	70.2	59.9	56.9	37.3	58.2	97.0
RAT-SQL	86.4	73.6	62.1	42.9	69.7	50.8	43.9	50.0	34.3	45.1	99.8
LGESQL	91.1	78.3	64.9	52.4	75.1	51.6	33.0	40.2	9.0	34.8	99.3
SmBoP	88.3	79.1	65.5	50.6	74.7	90.7	83.0	70.7	52.4	77.9	95.6
PICARD	89.9	83.4	66.1	44.0	75.5	95.6	85.4	67.8	50.6	79.3	98.1
CatSQL (w/o SC)	90.7	80.9	64.4	51.8	75.8	94.4	86.5	73.6	59.6	81.9	99.6
CatSQL (w/o PS)	86.7	70.6	59.2	50.0	69.2	89.9	72.0	69.5	50.6	72.4	100
CatSQL	**95.6**	**85.0**	**67.8**	**59.6**	**80.6**	**95.6**	**88.3**	**74.7**	**62.7**	**83.7**	**100**

Table 5: Execution accuracy on unpublished Spider test set.

Model Name	Acc_{ex}
SmBoP + GraPPa [35]	71.1
T5-3B + PICARD [37]	75.1
T5-SR (Anonymous)	75.2
RASAT + PICARD (Anonymous)	75.5
SHiP + PICARD (Anonymous)	76.6
Graphix-3B + PICARD (Anonymous)	77.6
CatSQL + GraPPa	**78.0**

generated SQL, and execute the SQL query over the underlying database and compare the results with a gold standard.

The NL2SQL community also uses another metrics called *logical form accuracy* (i.e., Acc_{lf}). This metrics considers a generated SQL query correct when it matches the gold standard exactly. In Spider, since the SQL literal may not appear in the natural language question, in its Acc_{lf} evaluation, it does not require the literal to be identical; therefore, when evaluating Acc_{lf} on Spider, a generated SQL query is considered correct if all other keywords and columns exactly match the gold standard but the literals may be wrong. Clearly this metrics is not as practical as execution accuracy, since (1) a practical NL2SQL application requires the generated literals to be correct; and (2) a syntactically different but semantically equivalent SQL query should be considered correct since it does not change the execution results. For Spider and WikiSQL benchmarks, we also provide the logical form accuracy results as a reference to the NLP literatures.

In addition, we report a new metrics called *exectuable rate*. This metrics reports the portion of generated SQL queries that can be successfully executed by the underlying database engine.

To evaluate the speed, we will evaluate different methods using the same hardware setup, and report the number of queries generated per second.

Baseline approaches. For rule-based systems, we compare NaLIR [26] and Templar [2]. For deep learning baselines, there have been extensive works evaluated against WikiSQL and Spider. To make a meaningful evaluation, we pick our baseline approaches based on the following criterion: (1) a method is reported to be among top of the leadership board by April 2022 and we have the code and the model to replicate the results; or (2) a method is widely compared against. All baselines and the pre-training models to achieve their best performance are summarized in Table 3.

Evaluation machine. All experiments are conducted on the a server machine equipped with an Intel Xeon Platinum 8163 CPU with 24 cores running at 2.5GHz, 128 GB memory, 2 TB of SSD, and one NVIDIA V100 GPU with 32 GB memory.

5.2 Cross-Domain Evaluation Results

Spider results. Our main evaluation results on Spider are presented in Table 4. From the table, we can observe that the two rule-based approaches, NaLIR and Templar almost fail on all test cases. This observation is the same as reported in [23]. The reason is that both of these approaches require tremendous human efforts to adapt to a new domain.

Among all deep learning-based approaches, we can observe that *CatSQL* is the best and significantly better than the second on almost all sub-tasks. Especially, on **Hard** and **Extra Hard**, *CatSQL* execution accuracy can achieve about 7 ~15 points higher accuracy than the second, i.e., PICARD. In total, *CatSQL* is 4.4 points better than PICARD on Acc_{ex}, which is a significant improvement considering that PICARD is only 1.4 points better than SmBoP, which is the 3rd on the current leadership board. Another important observation is that even without the *Semantics Correction*, *CatSQL* itself can achieve almost 2.6 points better execution accuracy than PICARD. It should be noted that PICARD uses a 8× larger model compared with *CatSQL*. This result demonstrates the benefits brought by our *CatSQL* sketch. Besides, we also evaluate the model without the parameter sharing (*CatSQL* w/o PS). It can be observed that the parameter sharing technique contributes 11.3 points to the overall accuracy, which demonstrates the effectiveness of the proposed architecture.

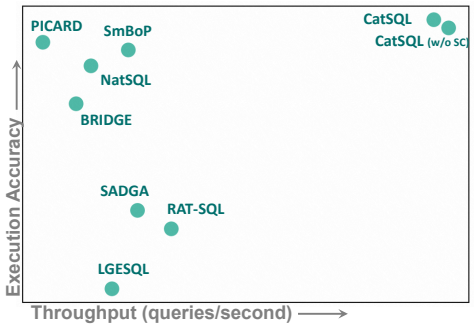

Figure 4: Execution accuracy versus throughput on several state-of-the-art NL2SQL approaches on the Spider dataset.

The advantage of CatSQL is also supported by the evaluation results on executable rate. From the result, all existing neural network models may generate un-executable queries while CatSQL is the first deep-learning-based model which achieves 100% executable rate. It should be noted that even without *Semantics Correction*, the executable rate of CatSQL is 99.6%, which significantly outperforms existing sketch-based models (e.g. RYANSQL and SmBoP), and sequence-to-sequence networks (e.g. PICARD). This result demonstrates the superiority brought by the newly introduced CatSQL architecture.

Another interesting observation is that, while our approach is not designed for logical form accuracy, *CatSQL* can also achieves a better Acc_{lf} than other approaches. In particular, LGESQL [8] and PICARD [37] are the 2nd and 3rd best on the Acc_{lf} leadership board, while *CatSQL* is 2 points better. We cannot evaluate the top-1 method called G³R since its model is not open-sourced. However, their reported Acc_{lf} on the `test` set is 77.2, which is much lower than ours. Thus we conclude that *CatSQL* is the new state-of-the-art on Spider's Acc_{lf} leadership board, although it is designed to optimize for Acc_{ex}.

We also submit our model to the Spider leaderboard on September. For a fair competition, the Spider official has not released the test set for public usage. At the time of writing, CatSQL achieved the first place on the overall leaderboard. Especially, *CatSQL* outperforms the last non-anonymous T5-3B + PICARD by a a significant and substantial margin of 2.9 points. The results on Table 5 demonstrate the effectiveness and generalization of *CatSQL*.

Last but not least, while our *CatSQL* is more accuracy, it also runs faster. Figure 4 demonstrates a diagram showing different approaches' execution accuracy versus throughput. From the figure, we can observe that *CatSQL* is better in both dimensions. Later in Section 5.4, we will show that *CatSQL*'s throughput is 63× larger than PICARD and 4.5× larger than SmBoP.

WikiSQL results. We present the WikiSQL results in Table 6. We can observe that our approach is better than all other baselines on all metrics. One main reason is that all baseline approaches train separate models for generating the **SELECT** clause and the **WHERE** clause. Different from all these approaches, our *CatSQL* technique uses the Column Action Template so that the model components generating the **SELECT** and **WHERE** clauses can share their parameters and contextual information, so as to improve the performance. All existing approaches use a technique called *Execution-Guided Decoding* [44]. We implement this technique as a special *Semantics Correction* rule, and thus we enjoy the benefit from it as well.

Table 6: Accuracy evaluation on WikiSQL.

Method	Dev		Test	
	Acc_{lf}	Acc_{ex}	Acc_{lf}	Acc_{ex}
SQLova	84.2	90.2	83.6	89.6
HydraNet	86.6	92.4	86.5	92.2
SDSQL	87.1	92.6	87.0	92.7
SeaD	87.6	92.9	87.5	93.0
CatSQL	**87.9**	**93.2**	**87.6**	**93.4**

5.3 Single-Domain Evaluation Results

We compare our approach against previous approaches on single-domain benchmarks as well. It should be noted that our *CatSQL* is trained only on the Spider training set for this evaluation. The results are reported in Table 7. Notice that our evaluation setup is the same as used in [23]. Therefore, we also include several deep learning baselines' results reported in [23].

We can observe that our *CatSQL* approach is superior than all other baselines on all benchmarks. That is, we are better than not only NaLIR and Templar, but also better than existing deep learning-based baselines. The only baseline close to *CatSQL* is NSP [21], which is only slightly lower than *CatSQL* on GeoQuery. The reason is that NSP uses manually annotated samples on the GeoQuery database for training while our *CatSQL* only uses Spider as the training dataset. All other baselines are far behind our *CatSQL* approach on all benchmarks.

We can observe that *CatSQL* achieves only 20% accuracy on IMDB. The major reason is that the join rules of most SQL statements in IMDB are not supported by *CatSQL* currently. This observation sheds light on the direction of future work. On all other benchmarks, *CatSQL*'s accuracy is above 60%, which demonstrates the adaptability of *CatSQL* to new database schema.

5.4 Throughput analysis

In this section, we study the throughput of our *CatSQL* prototype compared with baseline approaches. We mainly compare our system against Spider baselines and the two rule-based systems. We present the throughput results in Table 8. From the table, we can observe that our *CatSQL* system uses a relatively small model to achieve the state-of-the-art performance, and its overall latency is also the smallest. Our *CatSQL* system is almost 4× faster than SmBoP, and 18× faster than PICARD in terms of processing one query.

One interesting observation is that the latency of NaLIR and Templar, which does not use GPU at all is larger than *CatSQL*. The reason is because both these two systems requires visiting the entire database, which is expensive when the database size increases.

Our model has a smaller overall latency for two reaons. First, compared with most existing methods, *CatSQL* uses a relative smaller

Table 7: Execution accuracy on various single-domain benchmarks.

Benchmark	NaLIR [26]	Templar [2]	NSP [21]	SyntaxSQLNet [54]	GNN [6]	IRNet [16]	CatSQL (ours)
GeoQuery	0.36	0.0	70.0	49.64	38.21	60.36	**70.88**
Scholar	0.0	0.0	48.17	1.38	1.38	0.0	**60.79**
Restaurant	0.0	0.0	63.75	0.0	0.0	66.25	**71.80**
MAS	32.26	12.90	51.61	0.0	0.0	14.52	**60.71**
IMDB	0.0	11.90	16.67	0.0	2.38	14.29	**20.61**
YELP	4.88	9.76	0.0	0.0	0.0	0.0	**62.50**

Table 8: Inference throughput of different models. Latency indicates the overall latency of processing an entire query. Throughput is the number of queries that can be processed, when a maximum number of processes are launched to process the queries.

Method	#Para	Latency (ms)	Throughput
NaLIR	-	662.7	21.9
Templar	-	1140.8	9.5
RAT-SQL	449.4M	205.3	16.6
SmBoP	366.9M	489.3	11.3
LGESQL	375.4M	642.8	9.3
PICARD	2950.1M	2302.7	0.8
CatSQL	379.4M	**126.4**	**50.7**

Table 9: Latency of different components of *CatSQL*.

Process	Latency(ms)
CatSQL Embedding	42.6
CatSQL Decoding	63.7
CatSQL Semantics Correction	0.65

pretrained embedding model GraPPa, which requires less memory and computational resource. Second, at each decoding step, CatSQL generates a set of multiple slots in one `CAT`. This technique enhances the parallelization at inference and significantly reduces the time cost.

Since our *CatSQL* system launches a separate process to handle each query, and on our test machine, we can handle around 11 queries in parallel. Other systems using a larger model can handle a lower number of queries due to the limitation on GPU memory. This further amplifies the throughput advantage. In particular, compared with PICARD, our throughput is 63× larger than PICARD. Also, the throughput of NaLIR and Templar, which are not bounded by the GPU resources, is also worse than our approach, mainly due to the fact that their overall latency is larger.

We further give a detailed breakdown on each component of our system in Table 9. We can observe that the most time consuming component is the decoding network, which takes half of the time. Although the decoding network shares almost the same number of parameters with the GraPPa encoding network, the decoding phase needs to run a beam search, and thus its latency is larger than the encoding phase. Note that the decoding phase includes the literal handling algorithm, which takes less than 1 millisecond. Therefore, the literal handling's running time is a negligible compared with other components of the algorithm. Also, the *Semantics Correction* phase takes less than 1 millisecond, which is negligible compared with other components.

5.5 Case Study

We conduct an in-depth study of our *CatSQL* approach and existing approaches on the Spider dataset. We will first examine several cases that *CatSQL* can generate correct SQL queries while others cannot; then we will also discuss several error cases of *CatSQL*.

Correct Cases. We will show two complex cases that *CatSQL* succeeds. The first one is as follows.

Example 1. *Find the last name of the students who currently live in North Carolina but have not registered in any degree program.*

The ground truth is

```
SELECT T1.last_name
FROM Students AS T1 JOIN Addresses AS T2
  ON T1.current_address_id  =  T2.address_id
WHERE T2.state_province_county = 'NorthCarolina'
EXCEPT SELECT DISTINCT T3.last_name
FROM Students AS T3
  JOIN Student_Enrolment AS T4
  ON T3.student_id  =  T4.student_id
```

All other approaches failed on this case, because their performance decays significantly when handling nested queries. For example, PICARD will generate the following SQL query:

```
SELECT T1.last_name
FROM Students as T1
  JOIN Student_Enrolment as T2
  ON T1.current_address_id = T2.student_id
```

We can observe that the **SELECT** and **FROM** clauses are both correct, but the **WHERE** clause lacks a significant portion of information from the given natural language question.

Another example is as follows.

Example 2. *What are the first names of all players, and their average rankings?*

The gold SQL query is

```
SELECT avg(ranking), T1.first_name
FROM players AS T1 JOIN rankings AS T2
  ON T1.player_id = T2.player_id
GROUP BY T1.first_name
```

Systems such as PICARD will emit an almost correct SQL query, but the **GROUP BY** keyword is different (i.e. `T1.player_id`) We consider this case as a semantics violation, since we require the **GROUP BY** column(s) must appear in the **SELECT** clause as well. In this case, our *Semantics Correction* routine will correct this error case by replacing `T1.player_id` with `T1.first_name`.

Error Cases. The error cases show that our *CatSQL* still have rooms to improve. For example, the following question

Example 3. *What is the average and maximum capacities for all stadiums?*

CatSQL will translate it into

```
SELECT average, MAX(capacity) FROM stadium
```

while the gold answer is

```
SELECT AVG(capacity), MAX(capacity) FROM stadium
```

The column name `average` confuses the *CatSQL* generator to choose them to represent the description of "`average`" in the question. Such column names will confuse human users as well. A possible solution in real-world scene is to enrich the descriptive information of the column name (i.e., average_attendance).

Another class of errors happen when there is a complex usage of conjunction operators. For example, in the following example:

Example 4. *What are the names of properties that are either houses or apartments with more than 1 room?*

Our model generates

```
SELECT property_name
FROM properties
WHERE property_type_code = "House"
  AND room_count > 1
  OR property_type_code = "Apartment"
```

while the gold standard SQL is

```
SELECT property_name FROM Properties
WHERE property_type_code  =  "House"
UNION SELECT property_name
FROM Properties
WHERE property_type_code  =  "Apartment"
  AND room_count > 1
```

The conjunction relationship is translated into a wrong statement. Part of the reason is because our Column Action Template flatterns these conjunction relationship; and thus our approach is inherently weak on handling these cases. This problem could be possibly handled by either collecting more training data containing queries with complex conjunction relationships, or designing new architecture to predict logical operations explicitly. Both approaches are challenging to implement and require high-quality manual labels. We leave this issue as a further research to study.

6 RELATED WORK

NL2SQL problem is a long standing open problem lasting more than five decades since [14]. Earlier works are mostly rule-based. They develop searching-based techniques [4, 5] and parsing-based approaches [2, 22, 26, 36, 38, 50]. These approaches performs well on small single-domain datasets, and do not rely on the collection of a large training dataset; however, they typically do not scale well to a new domain for which the rules need to be manually re-designed.

With the recent advancements on deep neural network research and the availability of large training datasets [30, 55, 58], recent research exploits deep learning NL2SQL solutions and achieve the state-of-the-art results [12, 16, 17, 21, 23, 39, 43, 45, 51, 53, 58]. There are three lines of solutions. The first line takes a deep learning favor to consider the NL2SQL problem as a sequence-to-sequence translation problem [37, 58]. Such an approach typically struggles with generating syntactically legal SQL queries, and thus most of the efforts along this line are devoted to fixing the syntax issues. The second line of research employs a sequence-to-tree model to generate an abstract syntax tree along a predefined grammar to eliminate the syntax issue entirely [16, 35, 43], and demonstrates great promises towards tackling the problem.

Our work is along the third line of research, which takes a slot-filling-based solution [9, 48, 54]. In particular, these works design a sketch template with empty slots to fill, and the deep learning model only needs to predict the values to fill in the slot. This line of research typically enjoys great speed benefits since they do not need to waste computation resources to generate boilerplates such as keywords; so that all computation resources are devoted to extract meaningful information from the natural language to complements the SQL query. However, the accuracy of previous slot-filling approaches are typically low compared with the other two lines of works. We are the first work to demonstrate that a slot-filling approach can improve the state-of-the-art accuracy significantly.

All these methods are building on top of similar building blocks such as embedding network. While the state-of-the-art general purpose pretrained embedding network is based on pretrained language models[11], GraPPa [53] demonstrates that a specially designed embedding network for the NL2SQL task can significantly improves the performance. This line of research is orthorgonal and can benefit all deep learning-based NL2SQL solutions.

The collection of training datasets and benchmarks is also an important line of research. Existing works either manually build templates [3, 46] or employ a neural generators [15, 47] to automatically generate a large amount of data. This makes deep learning-based NL2SQL solution scalable. The most widely used benchmarks are WikiSQL [58] and Spider [55]. There are also some recent contributions which provide benchmarks for robustness evaluation, such as MT-Teql [30].

7 CONCLUSION

In this paper, we consider the NL2SQL problem and bridge existing rule-based solutions and deep learning-based solutions to achieve significant improvements in terms of both accuracy and throughput. First, we design *CatSQL*, a sketch-based NL2SQL solution, which is designed to be fast and accurate. Second, we develop the *Semantics Correction* technique, which is the first work to leverage database domain knowledge to improve the performance of deep learning-based NL2SQL algorithms. Our evaluation results demonstrate that our approach can significantly outperform previous approaches in a wide range of benchmarks. In particular, on the state-of-the-art cross-domain NL2SQL benchmark, Spider, our approach can improve the accuracy over previous state-of-the-art solution by 4 points on execution accuracy, while achieving an up-to 63× larger throughput.

REFERENCES

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint* (2014).

[2] Christopher Baik, HV Jagadish, and Yunyao Li. 2019. Bridging the Semantic Gap with SQL Query Logs in Natural Language Interfaces to Databases. In *ICDE*.

[3] Fuat Basik, Benjamin Hättasch, Amir Ilkhechi, Arif Usta, Shekar Ramaswamy, Prasetya Utama, Nathaniel Weir, Carsten Binnig, and Ugur Cetintemel. 2018. DBPal: A Learned NL-Interface for Databases. In *SIGMOD*.

[4] Hannah Bast and Elmar Haussmann. 2015. More accurate question answering on freebase. In *CIKM*.

[5] Sonia Bergamaschi, Francesco Guerra, Matteo Interlandi, Raquel Trillo-Lado, and Yannis Velegrakis. 2013. QUEST: a keyword search system for relational data based on semantic and machine learning techniques.

[6] Ben Bogin, Jonathan Berant, and Matt Gardner. 2019. Representing Schema Structure with Graph Neural Networks for Text-to-SQL Parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 4560–4565.

[7] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*. Springer, 177–186.

[8] Ruisheng Cao, Lu Chen, Zhi Chen, Yanbin Zhao, Su Zhu, and Kai Yu. 2021. LGESQL: Line Graph Enhanced Text-to-SQL Model with Mixed Local and Non-Local Relations. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 2541–2555.

[9] DongHyun Choi, Myeong Cheol Shin, EungGyun Kim, and Dong Ryeol Shin. 2021. Ryansql: Recursively applying sketch-based slot fillings for complex text-to-sql in cross-domain databases. *Computational Linguistics* 47, 2 (2021), 309–332.

[10] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. 2019. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In *International Conference on Learning Representations*.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL*.

[12] Li Dong and Mirella Lapata. 2016. Language to Logical Form with Neural Attention. In *ACL*.

[13] 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. In *Findings of the Association for Computational Linguistics: EMNLP 2021*. 2030–2042.

[14] Bert F Green Jr, Alice K Wolf, Carol Chomsky, and Kenneth Laughery. 1961. Baseball: an automatic question-answerer. In *western joint IRE-AIEE-ACM computer conference*.

[15] Daya Guo, Yibo Sun, Duyu Tang, Nan Duan, Jian Yin, Hong Chi, James Cao, Peng Chen, and Ming Zhou. 2018. Question Generation from SQL Queries Improves Neural Semantic Parsing. In *EMNLP*.

[16] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation. In *ACL*.

[17] Pengcheng He, Yi Mao, Kaushik Chakrabarti, and Weizhu Chen. 2019. X-SQL: reinforce schema representation with context. *arXiv preprint* (2019).

[18] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* (1997).

[19] Binyuan Hui, Xiang Shi, Ruiying Geng, Binhua Li, Yongbin Li, Jian Sun, and Xiaodan Zhu. 2021. Improving text-to-sql with schema dependency learning. *arXiv preprint arXiv:2103.04399* (2021).

[20] Wonseok Hwang, Jinyeung Yim, Seunghyun Park, and Minjoon Seo. 2019. A Comprehensive Exploration on WikiSQL with Table-Aware Word Contextualization. *arXiv preprint* (2019).

[21] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a Neural Semantic Parser from User Feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 963–973.

[22] Saehan Jo, Immanuel Trummer, Weicheng Yu, Xuezhi Wang, Cong Yu, Daniel Liu, and Niyati Mehta. 2019. Verifying text summaries of relational data sets. In *SIGMOD*.

[23] Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. 2020. Natural language to SQL: Where are we today? *Proceedings of the VLDB Endowment* 13, 10 (2020), 1737–1750.

[24] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[25] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 7871–7880.

[26] Fei Li and HV Jagadish. 2014. Constructing an interactive natural language interface for relational databases.

[27] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[28] Agnes Lydia and Sagayaraj Francis. 2019. Adagrad—an optimizer for stochastic gradient descent. *Int. J. Inf. Comput. Sci* 6, 5 (2019), 566–568.

[29] Qin Lyu, Kaushik Chakrabarti, Shobhit Hathi, Souvik Kundu, Jianwen Zhang, and Zheng Chen. 2020. Hybrid Ranking Network for Text-to-SQL. *arXiv preprint arXiv:2008.04759* (2020).

[30] Pingchuan Ma and Shuai Wang. 2021. MT-teql: evaluating and augmenting neural NLIDB on real-world linguistic and schema variations. *Proceedings of the VLDB Endowment* 15, 3 (2021), 569–582.

[31] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* 26 (2013).

[32] Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th international conference on Intelligent user interfaces*.

[33] 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* (2019).

[34] WG Redis. 2016. Redis. *URL: http://redis. io/topics/faq Accessed November* 17 (2016).

[35] Ohad Rubin and Jonathan Berant. 2021. SmBoP: Semi-autoregressive Bottom-up Semantic Parsing. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 311–324.

[36] Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R Mittal, and Fatma Özcan. 2016. ATHENA: an ontology-driven system for natural language querying over relational data stores.

[37] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 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.

[38] Jaydeep Sen, Chuan Lei, Abdul Quamar, Fatma Özcan, Vasilis Efthymiou, Ayushi Dalmia, Greg Stager, Ashish Mittal, Diptikalyan Saha, and Karthik Sankaranarayanan. 2020. ATHENA++ natural language querying for complex nested SQL queries. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2747–2759.

[39] Peng Shi, Patrick Ng, Zhiguo Wang, Henghui Zhu, Alexander Hanbo Li, Jun Wang, Cicero Nogueira dos Santos, and Bing Xiang. 2021. Learning Contextual Representations for Semantic Parsing with Generation-Augmented Pre-Training. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 13806–13814.

[40] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *NIPS*.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NIPS*.

[42] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *NIPS*.

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

[44] Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov, and Rishabh Singh. 2018. Robust text-to-sql generation with execution-guided decoding. *arXiv preprint arXiv:1807.03100* (2018).

[45] Wenlu Wang, Yingtao Tian, Haixun Wang, and Wei-Shinn Ku. 2020. A Natural Language Interface for Database: Achieving Transfer-learnability Using Adversarial Method for Question Understanding. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 97–108.

[46] Nathaniel Weir, Prasetya Utama, Alex Galakatos, Andrew Crotty, Amir Ilkhechi, Shekar Ramaswamy, Rohin Bhushan, Nadja Geisler, Benjamin Hättasch, Steffen Eger, et al. 2020. DBPal: A Fully Pluggable NL2SQL Training Pipeline. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2347–2361.

[47] Kun Xu, Lingfei Wu, Zhiguo Wang, Yansong Feng, and Vadim Sheinin. 2018. SQL-to-Text Generation with Graph-to-Sequence Model. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 931–936.

[48] Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint* (2017).

[49] Kuan Xuan, Yongbo Wang, Yongliang Wang, Zujie Wen, and Yang Dong. 2021. SeaD: End-to-end Text-to-SQL Generation with Schema-aware Denoising. *arXiv preprint arXiv:2105.07911* (2021).

[50] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *PACMPL*.

[51] Pengcheng Yin, Zhengdong Lu, Hang Li, and Ben Kao. 2016. Neural enquirer: learning to query tables in natural language. In *IJCAI*.

[52] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 440–450.

[53] Tao Yu, Chien-Sheng Wu, Xi Victoria Lin, Yi Chern Tan, Xinyi Yang, Dragomir Radev, Caiming Xiong, et al. 2021. GraPPa: Grammar-Augmented Pre-Training for Table Semantic Parsing. In *International Conference on Learning Representations*.

[54] Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. 2018. SyntaxSQLNet: Syntax Tree Networks for Complex and Cross-Domain Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*.

[55] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task.

[56] John M Zelle and Raymond J Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the national conference on artificial intelligence*. 1050–1055.

[57] Luke S Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: structured classification with probabilistic categorial grammars. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*.

[58] Victor Zhong, Caiming , and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint* (2017).