An optimized NL2SQL system for enterprise data mart

Kaiwen Dong✉[0000−0001−8244−9562], Kai Lu, Xin Xia, David Cieslak, and Nitesh V. Chawla[0000−0003−3932−5956]

Aunalytics, Aunalytics, South Bend IN 46545, USA
{kevin.dong,kai.lu,xin.xia,david.cieslak,nitesh.chawla}@aunalytics.com

Abstract. Natural language interfaces to databases is a growing field that enables end users to interact with relational databases without technical database skills. These interfaces solve the problem of synthesizing SQL queries based on natural language input from the user. There are considerable research interests around the topic but there are few systems to date that are deployed on top of an active enterprise data mart. We present our NL2SQL system designed for the banking sector, which can generate a SQL query from a user's natural language question. The system is comprised of the NL2SQL model we developed, as well as the data simulation and the adaptive feedback framework to continuously improve model performance. The architecture of this NL2SQL model is built on our research on WikiSQL data, which we extended to support multitable scenarios via our unique table expand process. The data simulation and the feedback loop help the model continuously adjust to linguistic variation introduced by the domain specific knowledge.

Keywords: Semantic Parsing · Natural language interface · Database · Language model

1 Introduction

Natural language interfaces to databases (NLIDB) [1] provide a way of interacting with relational databases by simply typing a question or Statement in natural language. This problem has been studied extensively, with early work in this field focusing on rule-based [4, 15, 21, 22] semantic parsing. The rule-based methods proved to be effective but lacked the ability to cover the linguistic variation and sophistication of end users. As deep neural networks have achieved state-of-the-art performance on numerous tasks around unstructured data [11, 23], the research interests of NLIDB have shifted to incorporate deep learning based approaches. Recent advances [8, 16, 26, 32] in the field have leveraged the release of large scale human-labeled datasets [33, 36] for model training and evaluation. However, there are few [34] that can be deployed on an actual enterprise data mart in production.

From a system and algorithmic perspective, NLIDB are difficult to develop and maintain as they require a substantial amount of expertise in machine learning methods, database architecture, microservices frameworks, infrastructure management, and DevOps practice. This type of application presents additional, non-obvious challenges for the machine learning practitioner. To begin with, an NLIDB faces an absolute dearth of available training data. Most ML methods require thousands – if not tens of thousands – of examples for reliable training. The most common bootstrap approach is to develop manually handcrafted example-and-label pairs; however, in this application building a suitable corpus upfront is extremely expensive and sometimes practically impossible. Should an NLIDB system complete initial training to satisfaction, it will face numerous ongoing operational issues. In some instances, new users will enter queries in unexpected ways and the system must accommodate feedback in order to continuously improve performance over time. Likewise, new versions of the underlying data model may incorporate new fields or tables, and it is critical that the system maintain expected performance on the original information while simultaneously integrating new database structures.

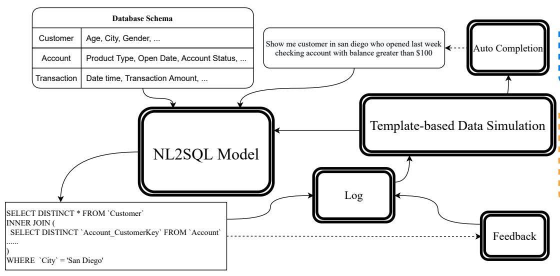
In this paper, we describe an NL2SQL system for the enterprise data mart [9] that can democratize access to relational databases for users without technical skills like SQL by allowing them to find meaningful insights and decisions with natural language. We will unfold how we developed this model in four sections. In the first section, we provide a functional description of the NL2SQL system including its web-based user interface, schema system and feedback logging mechanism. In the second section, based on the previous work of word contextualization [8] methods with a BERT-based encoder [3, 25], we discuss our design methodology and model architecture including its distinct subtasks and the novel table expand methodology we developed to support queries for a multi-table data mart. In the third, we discuss capability of being continuously optimized by a template-based data simulation which grows with the history of users' interactions with the model, which can generate data that improves performance on domain-specific language patterns and adapt to changes in the underlying data mart it is intended for. Fourth and finally, we offer the results of our benchmarking experiments that showed our model performing comparably to the other state-of-the-art models when trained on generic datasets from WikiSQL and Spider train, but significant improvements over that model when trained with our own template-simulated datasets.

2 Related Work

The release of the WikiSQL dataset [36] has raised interest in applying deep learning models to solve the text-to-SQL problem. Zhong et al. [36] introduced Seq2SQL, a sequence-to-sequence neural network. Xu et al. [32] proposed SQLNet structure using a sequence-to-set approach, which solves the order issue of the conditions in a SQL query. It also offers a column attention mechanism to identify the most relevant column for the natural language question. With transformer-based language models dominating most NLP tasks, Hwang et al. [8] leveraged the BERT [3] pretrained model and the stacked bidirectional LSTM [5] to construct a two-layer encoder and contextualize the natural language question with the headers. However, the research on WikiSQL is limited to one-table scenarios, due to the set structure of the dataset itself. Spider [33] proposes cross-domain text-to-SQL datasets across 200 databases, each database involving multiple tables with foreign keys. With the wide adoption of BERT in the encoding layer, Wang et al. [26] and Lin et al. [16] choose to concate-nate the question tokens along with the table and column name tokens, which are fed into the encoder to contextualize the word representation.

To provide a comprehensive solution for NLIDB, Li et al. [15] and Setlur et al. [21, 22] described systems with a rule-based parser and Dhamdhere et al. [4] discussed several implementation lessons and key design decisions for an industrial text-to-SQL tool. Zeng et al. [34] introduced a system consisting of a neural semantic parser, a question corrector, a SQL executor, and a response generator to tackle the task. For medical records information retrieval, Wang et al. [27] proposed a text-to-SQL system for relational databases at clinical centers. Data synthesis is essential when adapting the pretrained model to a specific domain; others have discussed several approaches [10, 27, 28, 35] to simulate the data consistent with the target domain where the system is being deployed.

3 NL2SQL System



Database Schema	
Customer	Age, City, Gender, ...
Account	Product Type, Open Date, Account Status, ...
Transaction	Date time, Transaction Amount, ...

```
SELECT DISTINCT * FROM `Customer`
INNER JOIN (
  SELECT DISTINCT `Account_CustomerKey` FROM `Account`
......
)
WHERE  `City` = 'San Diego'
```

Show me customer in san diego who opened last week checking account with balance greater than $100
Auto Completion
NL2SQL Model
Template-based Data Simulation
Log
Feedback

Fig. 1: Overview of the system workflow

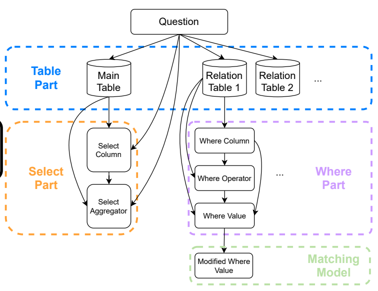


Fig. 2: The syntax-guided sketch and modules dependency

The design of our system is intended for practical industry application. fig. 1 shows the overview of the entire workflow. The system takes the user's question and the database schema as input and generates an executable query sent to a SQL engine to return the query result. While the user is typing the question, an auto-completion feature helps the user phrase their question. The NL2SQL model will then process the question and the schema. If the user is satisfied with the result, the query will be executed, otherwise the user can optionally submit feedback indicating the inaccuracy of the result.

The log is a place where we monitor the health of the model and seek opportunities to improve the model performance. A significant impact of the log is that it can guide the template writing for the data simulation process. The data simulation is the source for the training set of the NL2SQL model as well as the vocabulary for the auto-completion module. The template is updated continuously based on the logs generated by the users. This feedback loop helps us adjust the system to better adapt to users' language in a specific domain.

3.1 Question Textbox

The question text box is a standard text box, where the user inputs their question and submits it to the system. Hitting the submit button will launch a HTTP POST request to the server side and start the processing. There is also a collapsible schema viewer to remind the user of supported tables/columns in the database.

3.2 Schema

A relational database schema provides metadata like table names, column names, column types and foreign keys. Due to the abbreviated naming and blank space issues in the database world, the words of a column name are more likely to be concatenated together or linked by underscore and hyphen. This string format can cause problems when tokenizing. For instance, "AccountProductType" will be tokenized as "account", "#pro", "##du", "##ct", "##type" by the WordPiece [32] tokenizer, which leads to misinterpretations. Thus, our database schemas hold an optional human-readable alternative name for the tables and columns. The database developer can change these synonym names based on their need. This gives the end user more flexibility of how they shape their questions.

3.3 Auto Completion

Auto completion is another add-on feature embedded in the question textbox (section 3.1) Apart from the general advantage of helping users formulate their questions and reduce user-introduced typos, our design also guides the user to compose a question more likely to be recognized by the text-to-SQL parser. The engine we use is Elasticsearch [6]. We index the suggestion with the phrases (discussed in section 5.2) generated by the data simulation process, which will be further used as a source of training data.

3.4 Log

With the consent of the user, the logging system actively collects all incoming requests, the model execution log, and any feedback submitted by the user. The log can help quantify trends in the system usage and user satisfaction over time.

Log information is an essential source for continuous model improvement. Following a human-in-the-loop strategy, we pull the logs and analyze the system performance regularly. There are several aspects we look for in the logs: 1) the most asked-for tables and columns in the users' questions; 2) the questions that failed to be recognized by the system and are reported by users using the thumbs-down feedback mechanism; 3) the error messages; 4) the average latency of the request processing time.

The first two types of log data helps shape the template writing for data simulation, which provides a valuable channel to correct the bias of a language model like BERT [13, 24]. Log of error messages can capture the unexpected runtime errors. The average latency is also an important measurement indicating whether there is a need to scale up the service cluster.

4 Method

4.1 Problem Statement

We wanted to build an end-to-end model which takes a natural language question and database schemas (and potentially the data of the database if applying matching process section 4.5) as the input and generate SQL output. The query to be parsed is multi-table SQL without nested queries. To simplify the data structure of the SQL query, the SQL is converted into a logical form following the sketch style of SQLNET [32]. The sketch can ensure that the model always formulates the SQL query in a correct syntax. We have extended this sketch to support multi table samples. The query's component in this paper is always of the logical form.

Complexity always comes with flexibility in the SQL language. Even though the syntax-guided sketch [32] cannot completely cover the functionality of SQL, we still decided to employ it because of its standard structure. We are using this sketch to display the SQL query in an easier format for the user to understand. This can help the user make better decisions about whether the results from the system are desired.

The model is applied to the data model where foreign keys are predefined by a star schema [20]. Therefore, when generating the SQL query from the logical form, the database schema, instead of the NL2SQL model itself, will provide the necessary foreign keys to compose the SQL query for execution.

4.2 Model Overview

Following SQLOVA [8], the NL2SQL model is a sequence of sub-task classification models including SELECT column (sc), SELECT aggregation (agg), WHERE number (length of "conds", wn), WHERE column (wc), WHERE operator (wo) and WHERE value index (wvi). The tasks of WHERE value are tackled as a classification problem of locating the start and end tokens within the user's question as SQLOVA [8]. Besides the tasks above, we introduce 3 new tasks at the database level: main table (mt), relation table (rt), and relation number (length of "rt", rn), to parse the tables needed in the query.

There are 9 tasks in total with each model and these are formed in an encoder-decoder structure. All components share the same language model, BERT [3], as the first encoding layer, but the input can be different. Each module has its own bidirectional LSTM encoding layer. On top of the encoder, each task has its own classification layer. The tasks can be categorized into 3 parts based on their SQL clause: 1) Table part, including mt,rn,rt; 2) SELECT part, including sc,agg; 3) WHERE part, including wn,wc,wo,wvi. Categorizing modules into 3 parts gives us the advantage of only training a single part of modules when the dataset consists of both one-table and multi-table samples.

All the individual task module are similar to SQLOVA [8]. However, to accommodate multi-table requirements, we formulate a novel method called Table Expand to convert multiple tables per sample to one table in multiple samples.

4.3 Table Part

Similar to SQLOVA, Table part's input X_{Table} to the language model is composed of the tokenized natural language question and all the tokenized table names in the database. Question tokens and table name tokens are separated by a special token $[SEP]$. The tasks mt, rn and rt of the table part can be seen as the sc, wn and wc of the where part at the table level.

4.4 Table Expand

Once the relation tables $\hat{rn}$s of the database are selected, we can expand one sample to multiple samples by selected relational tables, each of which comprises a single selected table and the same question.

For instance, if two tables are selected after the Table Part module, then two inputs are fed to the BERT and the WHERE Part module. One includes the question and the column names from the first selected relation table, and the other includes the same question and the column names from the second selected relation table.

After the WHERE Part module processes all the expanded inputs and generates the WHERE number, WHERE column, WHERE Operator, and WHERE Value Index for each table, they are assembled back to be one sample again.

One sample of multiple tables can be expanded to several samples as below:

$$X_{Column} = [CLS], Q_1, \dots, Q_{L_Q}, [SEP], H_{1,1}, H_{1,2}, \dots, H_{1,L_1}, [SEP], \dots, [SEP], H_{N,1}, \dots, H_{N,L_N}, [SEP]; (\hat{rt}_j); j = 1, 2, \dots, \hat{rn} \tag{1}$$

where $\hat{rt}_j$ represents the index of the j-th selected relation table. $H_{i,L}$ is the L-th name token of the i-th column of table $\hat{rt}_j$. There are $\hat{rn}$ inputs in total.

The table expand can shrink the column search space for the model by splitting the selected relation tables to individual bins and limiting the number of columns to rank. Per each expanded input, the WHERE Number module will look for the number of WHERE columns $\hat{wn}_j$ from the current relation table

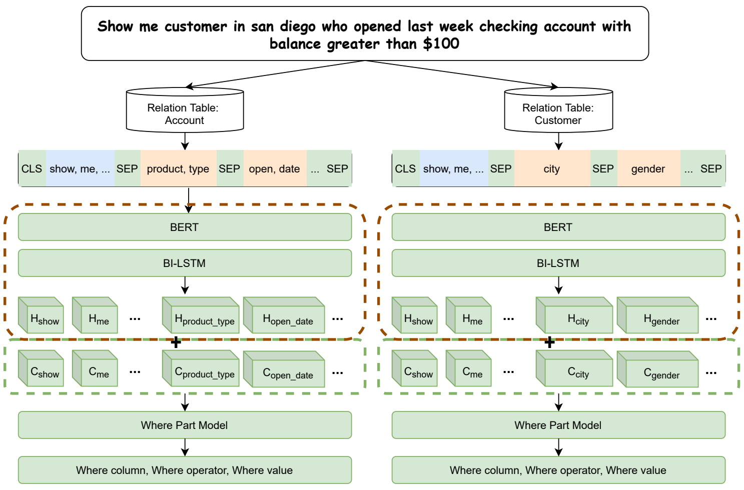

Show me customer in san diego who opened last week checking account with balance greater than $100
Relation Table: Account
Relation Table: Customer
CLS show, me, ... SEP product, type SEP open, date ... SEP
CLS show, me, ... SEP city SEP gender ... SEP
BERT
BI-LSTM
H_{show} H_{me} ... $H_{product\_type}$ H_{open_date} ...
H_{show} H_{me} ... H_{city} H_{gender} ...
C_{show} C_{me} ... $C_{product\_type}$ C_{open_date} ...
C_{show} C_{me} ... C_{city} C_{gender} ...
Where Part Model
Where column, Where operator, Where value

Fig. 3: Process of table expand and encoding

associated with the natural language question, instead of the total number of where conditions.

Another advantage of table expand is that it allows the schema to have more columns and more tokens for each column, despite limitation of BERT [3] input size. In the regime of the enterprise data mart, the size of a schema is usually larger compared to the Spider dataset [33], which is collected from public resources online. The schema of the database can impact the feasibility of the model and its performance. As most of the recent approaches to the Spider dataset share the idea of applying language models like BERT to contextualize the token sequences, it can potentially limit the number of columns to be recognized by the model. However, our method, by splitting the whole schema into individual tables when feeding the word sequence, can significantly increase the size of the schema supported by the model.

4.5 Where Value Matching

WHERE value can be obtained by locating the tokens in the substring of the original question with the start and end of WHERE value index. However, the extracted WHERE value usually does not match the exact cell value in the database, which can cause the query to be non-executable. For different data types, we set up different solutions to map the substring to the cell value in the database in table 1.

Categorical column FWe employ approximate string matching using Levenshtein distance [14, 18] to find the closest cell value in the predicted column compared to the extracted substring, which can help correct the syntactically similar string.

Table 1: Matching Process for different types of data

Date Type	Problem	Question	Substring	Table Cell
Categorical	Case or form doesn't match	Show me **mortgages**	Account_Type= "mortgages"	Account_Type= "Mortgage"
	No cell value present in question	Which customer **doesn't have** mobile bank?	HasMobileBank= "doesn't have"	HasMobileBank= "No"
		Customers with **dda**	ProductCategory = "dda"	ProductCategory = "Demand Deposit Account"
Datetime	The datetime format doesn't match	Accounts opened since **2018**	OpenDate> "2018"	OpenDate> "2018-01-01T00:00:00.000Z"
	Can't parse relative time expression	Accounts opened **this year**	OpenDate= "this year"	OpenDate≥ "2021-01-01T00:00:00.000Z" And OpenDate< "2022-01-01T00:00:00.000Z"
Numeric	The data type doesn't match	Accounts with balance more than **$100**	CurrentBalance > "$100"	CurrentBalance> 100.0

Because users usually don't type in the cell value explicitly in their question, it can raise the semantically close but syntactically different issue. We have an interim step when applying the approximate string matching. Instead of directly converting a substring to the cell value, we build a map dictionary between the cell value and the alternative strings. For instance, the binary value "Yes" or 1 will be mapped to a set of affirmation words like "with", "have", "has", "is", "are", while the binary value "No" or 0 will be mapped to a set of negation words like "without" , "don't", "doesn't", "isn't", "aren't". Cell value "dda" can be mapped to "Demand Deposit Account". When correcting the substring, we just find closest alternative value and then map it to the real cell value.

The building of the map dictionary is a rule-based iterative process. Even though we tried to apply other word representation technique like [19] to automate it, we found it is easier and more efficient to involve human-in-the-loop when solving these synonym wording requiring domain knowledge.

As the cell value needs to be preloaded, due to latency concerns, we will only string match for columns which have a relatively stable range of values. For example, the value set of column "Transaction_Type" is more consistent than "Transaction_Merchant_Name" over time.

In practice, we also set an empirical threshold to the distance depending on the sensitivity required for the matching process in case that the user indeed needs to query values not existing in the database.

Datetime column Parsing datetime text into structured data is a challenging problem. A question that includes a datetime value can be either absolute or relative. We utilize the Duckling [7] library to parse the extracted value to a formatted datetime interval following ISO8061 standard.

Numeric column A simple regular expression is applied to remove any non-number character in the substring, except the decimal point and the minus sign.

5 Template-based Data Simulation

Previous work in the healthcare domain [27] has dealt with specific medical terminology and the lack of questions to a SQL dataset in that domain. As the banking sector is the target domain to introduce our NL2SQL interface, optimization for this specific domain is required. In both domains, the cost of acquiring new training samples which pair natural language and queries is very expensive or difficult to acquire. The template-based data simulation provides a way to directly intervene with the model's capability for a specific domain. In production, the process can serve as a powerful tool to quickly correct the model when the user gives feedback about queries that could not be recognized. We also collect more language templates from the feedback and add them to our existing data simulation template.

The end goal of the data simulation is to generate pairs of the natural language questions and their corresponding queries. There are two steps to simulate the training samples, creating the query and creating the corresponding natural language question.

5.1 SQL query

The SQL query samples are generated based on the real databases where the NL2SQL interface is going to deploy. However, production data containing sensitive information must be substituted with dummy data.

The sample generation is a fill-in-slot process. After the number of conditions is randomly assigned, a permutation of wn will be picked of which the sum is the number of conditions. Then, all the other slots will be filled in randomly based on the database. mt is selected from the table names and the sc will be selected from table mt. agg can be no operation, AVG, COUNT, MAX, or MIN when sc is numerical column but can only be no operation or COUNT when sc is of categorical type. wc are selected from table rt based on the number wn. wo is selected from "=",">","<","! =" for the numerical column and "=", "! =" for string type. The wv will be sampled from the table cells of the database. A random state parameter of the generator is also required for purposes of reproducibility.

5.2 Natural Language Question

The template-based natural language generator can compose a question sentence corresponding to a SQL query. The template of the natural language needs to be customized for each data mart domain. Without applying any autonomous data augmentation on the natural language, we apply a rule-based simulation process for better control over what language expressions are generated. Thus, every word, except those coming from database values, is from the template we provide.

Each synthetic natural language question can be seen as a **Sentence** composed of **Phrases**. Each Phrase only holds the information of one condition in the **conds**, while a Sentence accounts for the whole query. The entire simulation process is to generate the Phrases and assemble them into a Sentence. The assumption of Phrase generation is that each column of the database can have its own expression style.

For each Phrase template, there are 2 major components: the column template and the value placeholder. The column template is composed of constant strings and placeholders for one column. The value placeholder will be substituted based on the elements of the condition in the query.

The column template needs to be set up for every field of all the tables in the database. If no template is specified for a column, a default one is used. The general workflow of composing a natural language question "Show me customers in San Diego who opened a checking account with balance greater than $100 last week" can be illustrated as follows:

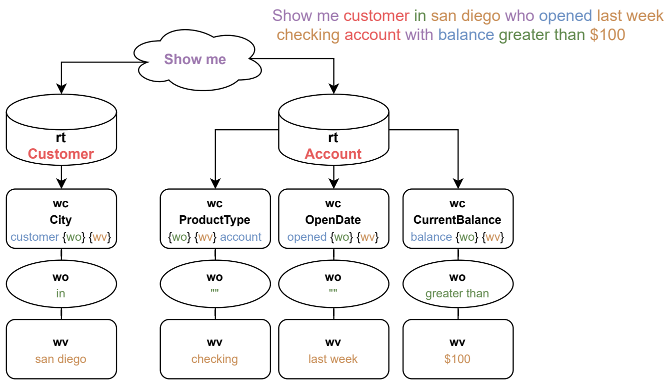


Fig. 4: Natural Language Generation from template-based Data Simulation

First, we need to generate each column phrase. The example contains four columns "City", "ProductType", "OpenDate", and "CurrentBalance" from "Account" and "Customer" tables. For "ProductType", it selects the "wo wv account" column template and then fills an empty string to the wo and the "checking" product type to the wv. For "CurrentBalance", the "balance wo $wv" template is chosen where wo and wv are filled with “greater than” and “100” accordingly. Similar approaches are applied for the remaining columns. After all conditions’ Phrases have been realized, we assemble them into a whole Sentence (see fig. 4).

5.3 Iterative Template Writing

In practice, the data simulation is an iterative process. Before exposing the model to any end user, we set up an initial version of the data simulation templates and trained the alpha model from it. Then, we conducted the first round of user acceptance testing (UAT) to gather real data from end users to the log, including the natural language question and possibly the expected SQL queries. This data is collected as our benchmark dataset for testing purpose. Instead of pouring this real data into the next round of training set directly, we firstly investigated those queries marked as “thumbs-down”, and figured out what language pattern, observed in user’s queries, can’t be generated based on template language. For instance, there was only “customer (in) (san diego)” generated by template initially, but we’ve seen queries from the users like “customer **(from)** (san diego)”, “**(san diego)** customer”, and “**members** (in) (san diego)”. Thus, we accommodate these language variations into the next round of data simulation by adding to templates.

Then, the model of next round will be trained entirely from the initial state but based on this new template. This process will be performed iteratively on certain cadence or on demand, which allows the model to evolve along with the utilization from users.

6 Experiment

6.1 Data

For the production model serving users, we build up our training dataset by the template-based data simulation process together with external data sources to improve the optimization to the focus domain for banking as well as the model’s robustness to linguistic variation.

We have two external data sources: WikiSQL [36] and Spider [33]. WikiSQL is limited to one-table scenarios so it can only be used to train the SELECT and WHERE parts of the entire model. Spider has more complex SQL queries like nested queries, which is not suitable for our model. In order to take advantage of the Spider train dataset, we only keep those queries that are compatible with our model. The criteria used to clean the external data sources include: 1) the total length of the concatenated input tokens to the BERT encoding layer tokenized by the WordPiece [31] tokenizer won’t exceed the allowed maximum length, which is 512; 2) the where value index can be parsed through CoreNLP tokenizer [17]; 3) the SQL query can be represented in logical form [32]. After cleanup, there are only 2286 samples from Spider train and 205 from Spider dev satisfying our needs. We denote them as $Spider_{Select}$ train and $Spider_{Select}$ dev. As the Spider test set is not publicly accessible, we use the $Spider_{Select}$ dev as the test set. $Spider_{Select}$ dev has 90 easy, 93 medium, 20 hard and 2 extra-hard question, defined by Spider. We only select 2000 samples from WikiSQL as $WikiSQL_{Select}$.

We also collected our own benchmark dataset from 3 rounds of UAT consisting of 289 samples, which can represent a wide range of user's input questions.

We used the $Spider_{Select}$ dev and our collected benchmark dataset as the test sets. WikiSQL is not usable because it only contains one-table samples.

6.2 Experiment Settings

The pretrained language model that we employ is the uncased BERT-base from Huggingface's library [30]. The entire dataset is composed of the synthesis pairs of NL questions and queries from the data simulation and samples from $WikiSQL_{Select}$ and $Spider_{Select}$ train set. The total number of samples are 7886. It is further separated into the train set and dev set. We use mini-batch size 1 and an early stop criterion on the dev dataset. The Adam optimizer [12] was applied. Other settings are the same as Hwang et al. [8].

The entire NL2SQL model consists of a sequence of 9 successive modules. The downstream tasks often rely on the result of the upstream tasks (see fig. 2). Thus, we employ teacher forcing [29] during the training phase for more efficiency.

We used a GPU to train the model, but CPU during inference. The UAT has confirmed the inference time for a request is acceptable to users, which is around 800ms on average. It allows us to deploy the system on clusters without GPU resources, which is more scalable.

6.3 Experiment Results

Table 2: Exact Match Accuracy Comparison on $Spider_{Select}$ Dev

$Spider_{Select}$ Dev (205 samples)	Easy	Medium	Hard
Bridge v2 + BERT [16]	0.89	0.53	0.25
NL2SQL by Data Simulation + $WikiSQL_{Select}$ + $Spider_{Select}$ train	0.71	0.52	0.4
NL2SQL by $WikiSQL_{Select}$ + $Spider_{Select}$ train	0.56	0.28	0.2

We use the Bridge model by Lin et al. [16], which is the top model on the Spider leaderboard at the time of writing, as a comparison for the logical form exact match accuracy. In table 2, our model trained on data simulation, $WikiSQL_{Select}$ and $Spider_{Select}$ train set underperform on the Easy queries of $Spider_{Select}$ Dev, compared to the Bridge model. However, our model catches up with the Bridge model on Medium and exceeds on Hard. Our model was exposed to a small portion of the original Spider train set while the Bridge was trained on the entire one. The Spider train and dev set shares the same databases and domains. When the expected SQL becomes more complex on Medium and Hard, the syntax-guided sketch of our model starts to show the advantage of composing longer and more difficult SQL queries.

We also trained a model on $WikiSQL_{Select}$ and $Spider_{Select}$ train set. It shows that a tool which can increase the size of training data like this simulation method can significantly improve the model performance on other domains even though the templates are not built for cross-domain data generation.

Table 3: Accuracy Comparison on our Banking benchmark

Banking benchmark (289 samples)	mt	rn	rt	sc	agg	wn	wc	wo	wv	Exact Match
Bridge v2 + BERT										0.01
NL2SQL by Data Simulation + $WikiSQL_{Select}$ + $Spider_{Select}$ train	0.82	0.99	0.73	0.74	0.99	0.7	0.67	0.66	0.62	0.45
NL2SQL by $WikiSQL_{Select}$ + $Spider_{Select}$ train	0.8	0.35	0.28	0.2	0.85	0.15	0.07	0.06	0.03	0.01

In table 3, the performance of our model is broken down into the different sub tasks introduced in section 4.2. As the output of Bridge model is SQL, we parse it into the syntax-guided sketch for evaluation, which can prevent grammatical errors. The Bridge model and our model trained without data simulation underperform on our banking benchmark dataset collected from the UAT. Our model with the data simulation process gains significant improvement to 45% exact match accuracy because of its better adaptation to the linguistic patterns in the domain. Note that the Matching Process is applied to both the Bridge and our model in this comparison.

7 Conclusion

This paper presents an optimized system for the enterprise data mart, including auto completion, neural semantics parser, cell value matching process and data simulation method. With the feedback loop and data simulation, our system has the potential to be applied on any enterprise data mart in different domains. It can remove a significant barrier to entry for querying databases for many participants without SQL knowledge, enabling non-technical users to perform analyses and make decisions.

Our system has been deployed and opened to users on our analytical database [2] for the banking sector. We are also working on presenting data visualization to users by interpreting their intention in the questions in order to provide graph visualizations in response to natural language questions. We will continue to explore the syntax-guided sketch to extend the coverage of questions our system can answer.

References

1. Androutsopoulos, I., Ritchie, G.D., Thanisch, P.: Natural language interfaces to databases - an introduction. CoRR **cmp-lg/9503016** (1995), http://arxiv.org/abs/cmp-lg/9503016
2. Aunalytics: Dayreak analytic database, https://www.aunalytics.com/products/daybreak/
3. Devlin, J., Chang, M., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. CoRR **abs/1810.04805** (2018), http://arxiv.org/abs/1810.04805
4. Dhamdhere, K., McCurley, K.S., Nahmias, R., Sundararajan, M., Yan, Q.: Analyza: Exploring data with conversation. In: Proceedings of the 22nd International Conference on Intelligent User Interfaces. p. 493–504. IUI '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3025171.3025227, https://doi.org/10.1145/3025171.3025227
5. Dong, L., Lapata, M.: Coarse-to-fine decoding for neural semantic parsing. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 731–742. Association for Computational Linguistics, Melbourne, Australia (Jul 2018). https://doi.org/10.18653/v1/P18-1068, https://www.aclweb.org/anthology/P18-1068
6. Elastic: Elasticsearch, https://www.elastic.co/enterprise-search
7. Facebook: Duckling, https://duckling.wit.ai/
8. Hwang, W., Yim, J., Park, S., Seo, M.: A comprehensive exploration on wikisql with table-aware word contextualization. CoRR **abs/1902.01069** (2019), http://arxiv.org/abs/1902.01069
9. Inmon, B.: Data mart does not equal data warehouse (1999)
10. Iyer, S., Konstas, I., Cheung, A., Krishnamurthy, J., Zettlemoyer, L.: 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). pp. 963–973. Association for Computational Linguistics, Vancouver, Canada (Jul 2017). https://doi.org/10.18653/v1/P17-1089, https://www.aclweb.org/anthology/P17-1089
11. Janai, J., Güney, F., Behl, A., Geiger, A.: Computer vision for autonomous vehicles: Problems, datasets and state of the art. Foundations and Trends® in Computer Graphics and Vision **12**(1–3), 1–308 (2020). https://doi.org/10.1561/0600000079, http://dx.doi.org/10.1561/0600000079
12. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: Bengio, Y., LeCun, Y. (eds.) 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings (2015), http://arxiv.org/abs/1412.6980
13. Kurita, K., Vyas, N., Pareek, A., Black, A.W., Tsvetkov, Y.: Measuring bias in contextualized word representations. In: Proceedings of the First Workshop on Gender Bias in Natural Language Processing. pp. 166–172. Association for Computational Linguistics, Florence, Italy (Aug 2019). https://doi.org/10.18653/v1/W19-3823, https://www.aclweb.org/anthology/W19-3823
14. Levenshtein, V.I.: Binary Codes Capable of Correcting Deletions, Insertions and Reversals. Soviet Physics Doklady **10**, 707 (Feb 1966)
15. Li, F., Jagadish, H.V.: Nalir: An interactive natural language interface for querying relational databases. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. p. 709–712. SIGMOD '14, Association for Computing Machinery, New York, NY, USA

(2014). https://doi.org/10.1145/2588555.2594519, https://doi.org/10.1145/2588555.2594519

16. Lin, X.V., Socher, R., Xiong, C.: Bridging textual and tabular data for cross-domain text-to-SQL semantic parsing. In: Findings of the Association for Computational Linguistics: EMNLP 2020. pp. 4870–4888. Association for Computational Linguistics, Online (Nov 2020). https://doi.org/10.18653/v1/2020.findings-emnlp.438, https://www.aclweb.org/anthology/2020.findings-emnlp.438
17. Manning, C., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S., McClosky, D.: The Stanford CoreNLP natural language processing toolkit. In: Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations. pp. 55–60. Association for Computational Linguistics, Baltimore, Maryland (Jun 2014). https://doi.org/10.3115/v1/P14-5010, https://www.aclweb.org/anthology/P14-5010
18. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. **33**(1), 31–88 (Mar 2001). https://doi.org/10.1145/375360.375365, https://doi.org/10.1145/375360.375365
19. Pennington, J., Socher, R., Manning, C.D.: Glove: Global vectors for word representation. In: Empirical Methods in Natural Language Processing (EMNLP). pp. 1532–1543 (2014), http://www.aclweb.org/anthology/D14-1162
20. Peterson, S.: Stars: A pattern language for query optimized schema (1994), http://c2.com/ppr/stars.html
21. Setlur, V., Battersby, S.E., Tory, M., Gossweiler, R., Chang, A.X.: Eviza: A natural language interface for visual analysis. In: Proceedings of the 29th Annual Symposium on User Interface Software and Technology. p. 365–377. UIST '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2984511.2984588, https://doi.org/10.1145/2984511.2984588
22. Setlur, V., Tory, M., Djalali, A.: Inferencing underspecified natural language utterances in visual analysis. In: Proceedings of the 24th International Conference on Intelligent User Interfaces. p. 40–51. IUI '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3301275.3302270, https://doi.org/10.1145/3301275.3302270
23. Shen, D., Wu, G., Suk, H.I.: Deep learning in medical image analysis. Annual Review of Biomedical Engineering **19**(1), 221–248 (2017). https://doi.org/10.1146/annurev-bioeng-071516-044442, https://doi.org/10.1146/annurev-bioeng-071516-044442, pMID: 28301734
24. Sun, T., Gaut, A., Tang, S., Huang, Y., ElSherief, M., Zhao, J., Mirza, D., Belding, E., Chang, K.W., Wang, W.Y.: Mitigating gender bias in natural language processing: Literature review. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics. pp. 1630–1640. Association for Computational Linguistics, Florence, Italy (Jul 2019). https://doi.org/10.18653/v1/P19-1159, https://www.aclweb.org/anthology/P19-1159
25. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. CoRR **abs/1706.03762** (2017), http://arxiv.org/abs/1706.03762
26. Wang, B., Shin, R., Liu, X., Polozov, O., Richardson, M.: RAT-SQL: relation-aware schema encoding and linking for text-to-sql parsers. CoRR **abs/1911.04942** (2019), http://arxiv.org/abs/1911.04942
27. Wang, P., Shi, T., Reddy, C.K.: Text-to-sql generation for question answering on electronic medical records. In: Huang, Y., King, I., Liu, T., van Steen, M.

(eds.) WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020. pp. 350–361. ACM / IW3C2 (2020). https://doi.org/10.1145/3366423.3380120, https://doi.org/10.1145/3366423.3380120

28. Weir, N., Utama, P., Galakatos, A., Crotty, A., Ilkhechi, A., Ramaswamy, S., Bhushan, R., Geisler, N., Hättasch, B., Eger, S., Cetintemel, U., Binnig, C.: Dbpal: A fully pluggable nl2sql training pipeline. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. p. 2347–2361. SIGMOD '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3318464.3380589, https://doi.org/10.1145/3318464.3380589
29. Williams, R.J., Zipser, D.: A learning algorithm for continually running fully recurrent neural networks. Neural Computation **1**(2), 270–280 (1989). https://doi.org/10.1162/neco.1989.1.2.270
30. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Brew, J.: Huggingface's transformers: State-of-the-art natural language processing. CoRR **abs/1910.03771** (2019), http://arxiv.org/abs/1910.03771
31. Wu, Y., Schuster, M., Chen, Z., Le, Q.V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, L., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., Dean, J.: Google's neural machine translation system: Bridging the gap between human and machine translation. CoRR **abs/1609.08144** (2016), http://arxiv.org/abs/1609.08144
32. Xu, X., Liu, C., Song, D.: Sqlnet: Generating structured queries from natural language without reinforcement learning. CoRR **abs/1711.04436** (2017), http://arxiv.org/abs/1711.04436
33. Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., Zhang, Z., Radev, D.R.: Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. CoRR **abs/1809.08887** (2018), http://arxiv.org/abs/1809.08887
34. Zeng, J., Lin, X.V., Hoi, S.C., Socher, R., Xiong, C., Lyu, M., King, I.: Photon: A robust cross-domain text-to-SQL system. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations. pp. 204–214. Association for Computational Linguistics, Online (Jul 2020). https://doi.org/10.18653/v1/2020.acl-demos.24, https://www.aclweb.org/anthology/2020.acl-demos.24
35. Zhong, V., Lewis, M., Wang, S.I., Zettlemoyer, L.: Grounded adaptation for zero-shot executable semantic parsing (2021)
36. Zhong, V., Xiong, C., Socher, R.: Seq2sql: Generating structured queries from natural language using reinforcement learning. CoRR **abs/1709.00103** (2017), http://arxiv.org/abs/1709.00103