Conversing with databases: Practical Natural Language Querying

Denis Kochedykov
kochedykov@gmail.com
JPMorgan ML CoE

Fenglin Yin
fenglinyin@gmail.com
JPMorgan ML CoE

Sreevidya Khatravath
sreevidya35@gmail.com
JPMorgan ML CoE

Abstract

In this work, we designed, developed and released in production DataQue – a hybrid NLQ (Natural Language Querying) system for conversational DB querying. We address multiple practical problems that are not accounted for in public Text-to-SQL solutions – numerous complex implied conditions in user questions, jargon and abbreviations, custom calculations, non-SQL operations, a need to inject all those into pipeline fast and to have guaranteed parsing results for demanding users, cold-start problem. The DataQue processing pipeline for Text-to-SQL translation consists of 10-15 model-based and rule-based components that allows to tightly control the processing.

1 Introduction

Large amount of companies' data are stored in relational databases – operational data, markets' data, clients data. These data are critical for decision making, however the most common channel for decision-makers to get a view of these data are (semi-) regular reports generated by data analysis professionals. Quick hypotheses validation is rarely, if ever, possible for majority of non-technical business stakeholders. Thus, one of the most valuable assets of a company – its data – often appears to be "locked" in the company's databases.

Another common interface to DBs for non-technical personnel are data dashboards. However, they have limitations – set of data views is usually fixed, not allowing for custom ones without writing structured queries; the UI of a dashboard tends to grow complex very quickly as business users ask developers to add more and more views, it becomes hard to navigate and hard to learn for new users; adding a new view to a dashboard requires developer work and takes time.

A solution to the problem are the so called Natural Language Querying systems with a Text-to-SQL models (Kumar et al., 2022), (Deng et al., 2022) or other text-to-query model at the core. NLQ allows non-technical users to formulate data requests as natural language questions. For example, *"show me sales last quarter by region"*. There is a lot of focus on NLQ recently in the industry, both large and specialized vendors offer NLQ solutions: Qlik, Tableau, PowerBI, IBM Watson, Amazon QuickSight, Google BigQuery, Tellius, Borealis, and dozens others. There are also multiple open-source models and frameworks for Text-to-SQL conversion trained on public datasets.

There are two critical factors that limit usefulness of most vendor and open-source NLQ solutions:

1. The questions that real business users ask don't look like the above "nice" data query – they are full of complex implied conditions, jargon, abbreviations, business rules, required custom calculations, non-SQL operations, etc. The solution should allow quickly and effortlessly inject such domain-specific business logic into the model pipeline and making it function in a deterministic "guaranteed" way according to this logic.

2. The requirement for an NLQ solution generally is to have close to 100% precision – so that user never unknowingly receives incorrect data in the response and makes decisions based on those incorrect data.

Let us use as a running example one of our internal applications – a database of daily profit and losses for thousands of financial trading desks in countries across the globe organized into a multi-level multi-dimensional taxonomy of businesses.

Consider a typical and a relatively simple user question in this usecase: *"yoy emea prime ytd pnl + forecast"*. First we can notice it barely resembles proper English. Trading floor language is often very compressed and full of jargon. Public NLQ

Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: Industry Track, pages 372–379
December 6-10, 2023 ©2023 Association for Computational Linguistics

systems trained on proper English inputs would be mostly useless on such questions. Unpacking this question gives us *"over the past 5 years, in every year, take same dates range as between the first day of the year and the current date, select P&L for Prime Finance trading desks in EMEA region countries and concatenate these values with full year forecasted P&L values"*. This is something that public NLQ systems could parse, but, unfortunately, real business users never formulate questions like this. One could think it's possible to solve this with a proper query expansion logic – i.e. making all the above substitutions in the question itself. However, the business logic in many cases depends on the 1) the presence or absence of certain entities, e.g. if "moving avg" is mentioned – this implies different default date range 2) on the value of entities, e.g. implied conditions might be different for different trading desks and 3) on the DB table that user is asking about. So even to expand the question, we first need to parse it and convert to SQL, thus the text-to-sql model should work on these "short" questions. Then the business logic is applied on the *parsed* question and at *different stages* of parsing – some at NER/NED stage, some at intent understanding, some at SQL query construction, etc. See section 4.1 for statistics of sample user questions.

In the section 2 we review some related work, in section 3 we review the system architecture and in section 4 we provide some testing results.

2 Related work

Text-to-SQL translation has become an active research topic for a number of years. Solutions at early stage were more domain-specific, and often adopted rule-based approaches (Stratica et al., 2005). In recent years, several public data sets, including WikiSQL (Zhong et al., 2017), Spider (Yu et al., 2019b), and CoSQL (Yu et al., 2019a), became available, which contain hundreds of databases from diverse domains. Effort is shifting towards building domain-agnostic generic model with deep neural networks. The structure of neural networks model evolved from general purpose sequence-to-sequence model, to sequence-to-sketch model (Xu et al., 2017), and normally employs encoder and decoder structure. On the encoder side, techniques such as relation-aware schema encoding (Wang et al., 2021a) has been developed to help link tokens in question to database schema. On the decoder side, different techniques around constrained decoding, such as PICARD (Scholak et al., 2021), has been developed to improve decoder performance. Instead of decoding directly into SQL statement, it's easier to decode into an intermediate representation of SQL. Different ways of generating intermediate representation of SQL have been proposed in RAT-SQL (Wang et al., 2021a), Syntax-SQL (Yu et al., 2018), NatSQL (Gan et al., 2021). Another option is to decode into an abstract syntax tree (AST) (Yin and Neubig, 2017).

Inspired by sequence-to-sketch model and the AST approach, we designed and developed an NLU parsing pipeline and slot-based SQL generation engine, as well as a custom AST that can be used for SQL generation, as well as for modification of SQL based on business rules. As training of text-to-SQL model requires large amount of labelled data, various data augmentation methods have been proposed and tested (Wang et al., 2021b). We also developed data augmentation tool using techniques, such as entity swapping, and paraphrasing. Most recent advancements in the field are around leveraging capability of generative large language models for Text-to-SQL translation through prompt engineering and in context learning (Pourreza and Rafiei, 2023).

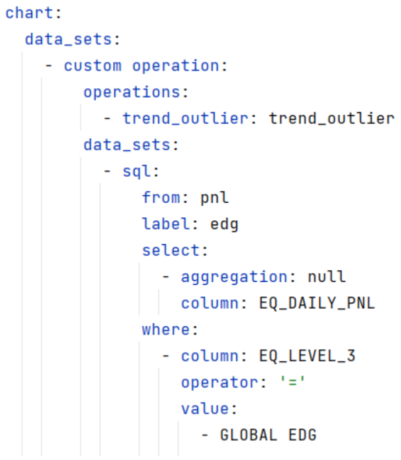
3 Methodology

3.1 AST

We designed custom Abstract Syntax Tree (AST) representation for complex queries combining SQL and non-SQL elements in a tree-like structured object. The AST is designed to fit sketch-based and slot-based query generation process, as well as business rule driven expansion process (see below). Figure 1 shows an example AST from query *"edg pnl outliers plot"*.

3.2 Pipeline

The pipeline of DataQue is presented at high level in figure 2. Components such as NER, NED, AST constructor, and AST expansion component are core/critical components. We also have utility components such as intent classifier, OOD classifier, follow-on classifier, and dialog state tracker. We will describe the components one by one.



```
chart:
  data_sets:
    - custom operation:
        operations:
          - trend_outlier: trend_outlier
        data_sets:
          - sql:
              from: pnl
              label: edg
              select:
                - aggregation: null
                  column: EQ_DAILY_PNL
              where:
                - column: EQ_LEVEL_3
                  operator: '='
                  value:
                    - GLOBAL EDG
```

Figure 1: Example AST serialized into YAML format.

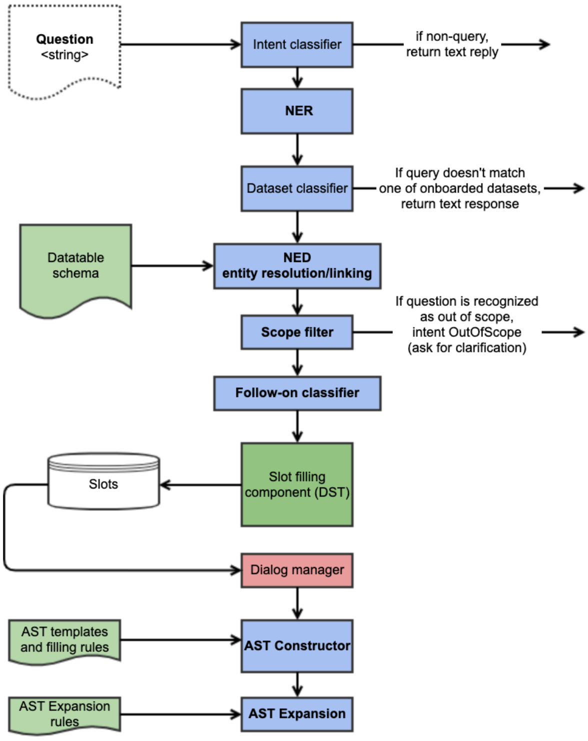


Figure 2: High-level schematic pipeline of DataQue conversational NLQ solution.

Intent classifier

What: Classifies if the input user utterance is one of the conversational intents (hi/bye/thanks/frustration/etc) or a data question.

Why: Not all the inputs into the system are proper data questions, we don't want to try to translate to SQL an input *"hi"*.

How: A standard transformer-based text classifier; we crowd-sourced some examples of conversational intent utterances and we used the test examples of data questions as training examples for intent classifier.

Exact pattern matching component

What: Matches user utterance against list of predefined regular expressions and, if a match is found – retrieves corresponding SQL translation.

Why: For some simple queries users need a guaranteed result, e.g. a query like *"pnl"* – there are too many implied conditions in it like date range, aggregation, geography, etc, to pass it through pipeline.

How: A list of regular expressions and corresponding AST translations.

NER (Named Entities Recognition) component

What: Identifies entities in the input utterance - trading desk names, countries, dates and ranges (*"now"*, *"last week"*, etc), numerical columns (like *"pnl"*,), analytics operations (*"moving avg"*, *"std"*, *"runrate"*, etc), aggregations (*"yoy"*, *"daily"*, etc), postprocessing operations (*"chart"*, *"outliers"*, etc), financial products (*"cash"*, *"derivs"*, *"brokerage"*, etc), and multiple others.

Why: There are multiple reasons to have NER and NED (Named Entities Disambiguation) as separate explicit components rather than to include it implicitly in the way the tokens are encoded in a transformer-based model's attention, as done, e.g in (Wang et al., 2021a). Mainly, because many of the business rules for expansion of AST depend on the set of entities identified in the question. For example, when user mentions *"runrate"* or *"daily avg"*, it triggers multiple business rules AST expansions: 1) the formula for the calculation is included into the AST using only the business days for averaging 2) if date range is not specified, "year to date" is used (and this itself depends on presence of another entity – date_range). It's very hard if possible at all to specify this sort of logic in the attention-based encoding of tokens. Secondary reason is that model-based tokens attribution requires sizable training data and, because data for practical NLQ applications are quite different from public datasets – these data need to be collected from scratch, we have a cold-start problem.

How: We use 3 complementary NER components: 1) regular-expression-based and dictionary-based extraction 2) standard transformer-based NER model trained on a small set of domain-specific data 3) standard NLP libraries like Spacy and Duckling for standard entities like dates, money, etc.

Table classifier

What: Uses the wording of the question and extracted entities to recognize which DB tables the question refers to, e.g. P&L table, market data table, etc

Why: We need table classifier to feed into the downstream components: 1) we need to determine which table(s) NED will link the extracted entities, 2) the business rules for AST expansion are often different for different DB tables, so we need to understand how to expand the parsed question's AST.

How: Standard transformer-based classifier using the wording of the questions and the extracted entities as inputs. We use the small set of table specific questions to train the classifier.

NED (Named Entities Disambiguation)

What: Resolves the face value of an entity extracted from the question: 1) standard resolution for dates (*"yesterday"* → last business day → date of the last week's Friday), money, etc 2) for entities that refer to the DB schema – link them either to a table name or a name of a column in a table or an individual value in one of the columns in a table.

Why: To be able to form an SQL query, we need to resolve face-value of extracted entities to their numerical values or canonical values from the DB table(s).

How: For resolving standard entities we use standard NLP packages like Spacy and Duckling. For linking entities to DB schema, we use a combination of approaches: 1) regular-expression-based and dictionary-based matching, e.g. we have a list of all typical ways how business-users refer to certain desks 2) fuzzy matching based on the string distance. The first approach is also useful when (often) a single entity cannot be linked to a single value in the DB table – users refer to multiple values by one short abbreviation, e.g. they refer to a group of trading desks rather than a single desk. In such cases it's pretty straightforward to add the corresponding shortcut to a dictionary-based linking. The second approach is useful for linking entities to values in the table with many unique values, e.g. when a user refers to some trading desk by name but not use a proper name and rather use some sort of abbreviation. The DB is scanned regularly, all columns names are extracted and all unique values in the table for all non-numerical columns are extracted and used for linking of entities based on string distance.

OOD (Out-Of-Domain) classifier

What: A binary classifier that recognizes if the user's question is in-scope for the system or not.

Why: One critical requirement for a practical NLQ system is that user never unknowingly receives data that doesn't match the the user's question. This means the system needs to produce a confidence score for each text-to-sql translation and, if the system is not highly confident in the translation, it needs to fallback to a clarification/disambiguation question to the user. For example, when user asks *"corr b/w pnl and snp"* meaning *"compute correlation between daily P&L and the S&P500 market index this year to date"* – although all the entities might be recognized, this is not something the system can do and it should respond accordingly. The second important, but conflicting requirement is that the system needs to be highly controllable, easily extensible with complex business logic. This means that significant portions of the system needs to be rule based. However rule-based components don't produce confidence scores. To address this, we run a classifier on the parsing results that says if the question is likely to be in-scope question and the parsing result is likely to be correct or otherwise. The classifier can produce a confidence score that can be thresholded and, for low confidence, the system resorts to fallback clarification with the user.

How: We train a classifier using a variety of features – the wording of the question, the # of entities recognized in the question, % of tokens in the question that are not entities, if there are duplicate entities recognized in the question (e.g. 2 date ranges), confidence of model-based NER components, # of entities that were not linked to DB schema and other. We train the classifier using a small set of domain-specific parsing examples as the positive class and we generate examples for the negative class in 2 ways 1) by running random out of domain sentences through the system 2) by corrupting positive in-domain examples in various ways making them non-parsable (randomly introducing unsupported operations, tables, and analytic functions).

Follow-on classifier

What: A component that determines which user questions are follow-on questions to the user's previous question and which are genuinely new questions.

Why: Users' expectation is that they can refine their question in a conversational manner and "drill down" exploring their DBs. The logic for processing the question depends on whether it's a new question / topic change or a follow-on. For example, the conversation **user**: *"emea pnl last year"* → **nlq**: (...) → **user**: *"+asia"*; → **nlq**: (...) → **user**: *"plot"* → **nlq**: (...) → user: *"prime pnl last week"*; here

- the second utterance *"+asia"* is a follow-on question meaning *"concatenate to the previous result a column with P&L from APAC region countries for the same dates and also compute the total"* and it only can be understood in the context of the previous question,
- the third question is also a follow-on meaning *"plot the above 2 series"* and
- the last question is a topic-change.

How: We use a rule-based binary classifier component based on features like wording, length, number of entities in the question and status of memory slots.

DST (Dialog State Tracker)

What: Component that fills the memory slots with entities and intents extracted from previous questions. Slots' values then used for constructing the AST for the question. The function of this component is to decide – given the entities extracted and resolved from the current user utterance and the current state of memory slots – which slots do we update with the new values, which slots we keep as-is and which we reset.

Why: Same as the above follow-on classifier, the DST component allows to carry context from question to question in a conversational NLQ. Simplistically, we could rely only on the follow-on classifier: if the question is a follow-on – put all entities from the question into corresponding slots and preserve other slots values, if the question is a topic change – reset all memory slots and put new values. However, in practice, different slots can have their own idiosyncratic rules for when to keep them and when to reset. There are also follow-ons like *"i need asia, not global"*, where though two "region" entities are extracted, only one need to be put in the corresponding memory slot.

How: We use a simple rule-based 3-class classifier component that determines for each memory slot if it needs to be filled/reset/kept. Inputs into the classifier are the follow-on classifier output, the extracted entities, the current values of the slot.

AST (Abstract Syntax Tree) Constructor

What: Component that generates initial AST based on parsing results of previous components.

Why: Need to convert natural language query into instructions on how to extract, process and render data, so that downstream AST executor can act on.

How: We have a set of AST templates defined in YAML format. We populate the templates based on extracted and resolved/linked entities - populate SELECT, FROM, WHERE, GROUPBY, aggregation, in the SQL queries in AST, add post-processing and custom analytics nodes in the AST based on templates.

AST expansion component

What: Initial AST needs to be expanded based on implied default conditions and business rules.

Why: The AST that can be constructed from the user question is often only partially filled. There are business rules on 1) how to fill in default value when it's not available in user query; 2) how to concatenate additional data for comparison with requested data; 3) how to present result tables in certain formats. AST expansion adds all these extra conditions, data, calculations and formatting.

How: Rules are captured in YAML file, each rule has triggering condition and corresponding action. E.g: if date range is missing and analytics operation present → default to year-to-date; if date range is this year → pull last five years to compare with; if multiple group-by conditions → pivot table to both row and column.

4 Training and evaluation

4.1 Data

Because the trading language and the structure of the questions are very domain-specific, we cannot leverage public datasets and had to collect data internally. It's not feasible to collect data directly from traders, so we collected data in 2 steps: 1) generated synthetic data from template questions and leveraged internal annotation team to write more labelled examples; 2) after the system was released into production, we run it for a short period of time and collect example questions asked by real trading desks users.

The datasets we used in training and testing:

#1 **Synthetic questions**: we used ≈10 template question structures and generated ≈20k synthetic questions by substituting various admissible combinations of entities' values

#2 **Annotated questions for NER/NED training**: we used internal annotation team to write ≈300 example questions and annotate them with entities and resolutions – linking to DB schema or resolved value of non-DB entities.

#3 **Annotated questions for NER/NED testing**: same source as above, ≈200 questions

#4 **Examples of non-business utterances**: annotation team wrote ≈200 examples of non-business utterances: greeting, affirm, deny, thanks, chitchat, etc

#5 **Production run questions**: ≈300 business questions sampled from initial period of production run with real users.

For the questions in dataset #5, some characteristics:

- Average length of questions: 31 characters; 6 tokens.
- Average length of AST SQL: 22 tokens.
- Average length of expanded AST SQL: 33 tokens.
- The SQL's are single table queries, with about 15% having custom non-SQL analytic or comparison operation.
- Average number of special entities (jargon, abbreviation, etc) in each question is 2.

4.2 System training and evaluation

We evaluated individual components of the system and the whole system accuracy on a production flow dataset, we present here only some key components evaluation.

NER

As described in the section 3 , the component consists of rule-based and model-based parts. For the model-based part we used the DIET (Dual Intent Entity Transformer) model (Bunk et al., 2020) with 12 transformer blocks, batch size 128, trained for 75 epochs. Inputs into the model were 768-dim embeddings by Bert-based featurizer. We use combination of datasets #1 and # 2 above for training and dataset #3 for testing the NER model. Weighted average F1 score across all entities types is 97%.

NED

As discussed in section 3, the component is based on rules, dictionaries and string distance matching logic, there are no trainable parameters. The dataset #3 is used for testing the accuracy of NED. The weighted average F1 score across all entities types is 96%.

Overall system execution accuracy

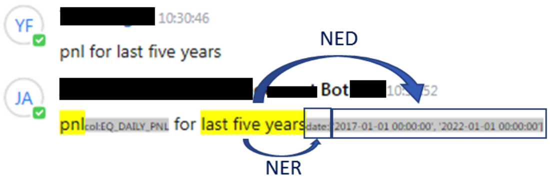
We use dataset #5 – production data – for testing overall system performance. We measure the fraction of queries in the test set giving EM (Exact Match) of the data returned by the whole system and the expected data to be returned. This metric measures the accuracy of all components in the system together - rule-based, model-based and purely engineering like query execution and even the accuracy of the data in DB. The EM score is 88%.

5 Discussion

The overall accuracy of the system is relatively high at 88%, however, production requirement is that users never unknowingly get incorrect data from the system. Error analysis indicated that the 12% of cases when the system didn't return the expected data are split approximately 30%/70% between 1) system returning *some* data and 2) system returning a text message (e.g, Out-of-Domain classifier recognized the question as OOD or intent classifier mis-recognized the intent). This means, in 12%·30% ≈ 4% cases, the system returns data that doesn't match the user's question. Further error analysis indicated that in almost all of these cases, the system was not able to recognize or resolve one of the entities that user has mentioned. To address these situations, we take "human in the loop" path and show the user the parsing results in a simplified form, see fig. 3. This allows users to recognize when the system incorrectly parsed their question and returned them not the data they asked.

6 Conclusions

We designed, developed and deployed in production a hybrid conversational NLQ system for several real-world usecases in a large international financial institution. The approach allows to address multiple conflicting practical requirements – custom domain language with jargon and abbreviations, numerous complex implied conditions



YF [redacted] 10:30:46

pnl for last five years

JA [redacted] [redacted] Bot [redacted] 10:[illegible]52

pnl col:EQ_DAILY_PNL for last five years date: '2017-01-01 00:00:00', '2022-01-01 00:00:00']

Figure 3: Example "explainability" part of the system's output

in the users' questions, need to incorporate new business-rules into the model pipeline quickly as user feedback comes and have a guaranteed behavior of the system, high accuracy and interpretability of the results. Vendor solutions or open-source end-to-end Text2SQL models do not allow for required level of customization and controllability for heavily domain-specific applications. We believe, our hybrid framework strikes a good practical balance between leveraging pre-trained language understanding models and rules.

Limitations

One limitation of the approach is the other side of its strength and is in line with the usual dichotomy "rules – high precision, low recall, statistical models – lower precision, but higher recall". The proposed approach has a sizeable rule-based part so, while it is highly controllable and addresses the cold-start problem, it is not as robust to variations in the questions, e.g. when understanding the question requires reasoning or common sense knowledge. Similarly, it's not as robust to variations in the dialog scenarios outside "question+follow-ons", to users explicitly referring to something in the previous questions or previous answers and other. The conscious design choice we made that for such variations the system defaults to error messages, rather than running a risk of providing incorrect answers.

Another downside of the approach is also the consequence of using rule-based components in the pipeline – the set of rules need to be maintained clean and up-to-date, e.g. the dictionaries, the AST expansion rules and other. This requires some effort from the developer team running the system in production.

Here are a few examples that current solution of DataQue failed.

1. *"How well have I done in this month as compared to last year"* – failed to understand intent.
2. *"ytd edg pnl excluding corps and converts"* – failed to understand "excluding" operation.
3. *"split per stripe"* – failed to understand the group-by condition stated in this follow-on question.

Future Work

The initial development of the system had the cold-start problem – business users could not be asked to write any significant amount of queries and explanations and professional annotators could not representatively capture very domain-specific lingo. We addressed the problem by decomposing the NLQ pipeline into small tasks and mixing rule-based and model-based components. Following the production release of the system, one immediate direction of improvement is collecting more users queries from system usage in order to create larger training/testing datasets for components.

Another immediate direction of work is leveraging Large Language Models in the pipeline. LLMs excel in natural language understanding and have demonstrated proficiency in code generation tasks, making them seemingly well-suited for NLQ solutions. However, for a domain-specific NLQ task, initial study indicates that out-of-the-box LLMs actually struggle to perform. One challenge is the consistency of the LLMs responses – it far doesn't fit the requirement of "close to 100% precision" in business-critical domains. Another challenge is latency. To make responses more consistent and address all the custom requirements described in this work, several additional components still needed in the pipeline. If LLMs are used, multiple calls are needed and this leads to a significant overall latency, with complex queries taking over 30-60 seconds to process. In contrast, the solution presented in this work achieves a latency of less than one second. It is likely that the ultimate solution will be a combination of the pipeline presented in this work with some LLM components.

Ethics Statement

All the work done and discussed in this paper meets and upholds the ACL Code of Ethics.

References

Tanja Bunk, Daksh Varshneya, Vladimir Vlasov, and Alan Nichol. 2020. Diet: Lightweight language understanding for dialogue systems.

Naihao Deng, Yulong Chen, and Yue Zhang. 2022. Recent advances in text-to-SQL: A survey of what we have and what we expect. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 2166–2187, Gyeongju, Republic of Korea. International Committee on Computational Linguistics.

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

Ayush Kumar, Parth Nagarkar, Prabhav Nalhe, and Sanjeev Vijayakumar. 2022. Deep learning driven natural languages text to sql query conversion: A survey.

Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of text-to-sql with self-correction.

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

Niculae Stratica, Leila Kosseim, and Bipin C Desai. 2005. Using semantic templates for a natural language interface to the cindi virtual library.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2021a. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers.

Bailin Wang, Wenpeng Yin, Xi Victoria Lin, and Caiming Xiong. 2021b. Learning to synthesize data for semantic parsing. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2760–2766, Online. Association for Computational Linguistics.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning.

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)*, pages 440–450, Vancouver, Canada. Association for Computational Linguistics.

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.

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

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2019b. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103.