TypeSQL: Knowledge-based Type-Aware Neural Text-to-SQL Generation

Tao Yu
Yale University
tao.yu@yale.edu

Zifan Li
Yale University
zifan.li@yale.edu

Zilin Zhang
Yale University
zilin.zhang@yale.edu

Rui Zhang
Yale University
r.zhang@yale.edu

Dragomir Radev
Yale University
dragomir.radev@yale.edu

Abstract

Interacting with relational databases through natural language helps users of any background easily query and analyze a vast amount of data. This requires a system that understands users' questions and converts them to SQL queries automatically. In this paper we present a novel approach, TYPESQL, which views this problem as a slot filling task. Additionally, TYPESQL utilizes type information to better understand rare entities and numbers in natural language questions. We test this idea on the WikiSQL dataset and outperform the prior state-of-the-art by 5.5% in much less time. We also show that accessing the content of databases can significantly improve the performance when users' queries are not well-formed. TYPESQL gets 82.6% accuracy, a 17.5% absolute improvement compared to the previous content-sensitive model.

1 Introduction

Building natural language interfaces to relational databases is an important and challenging problem (Li and Jagadish, 2014; Pasupat and Liang, 2015; Yin et al., 2016; Zhong et al., 2017; Yaghmazadeh et al., 2017; Xu et al., 2017; Wang et al., 2017a). It requires a system that is able to understand natural language questions and generate corresponding SQL queries. In this paper, we consider the WikiSQL task proposed by Zhong et al. (2017), a large scale benchmark dataset for the text-to-SQL problem. Given a natural language question for a table and the table's schema, the system needs to produce a SQL query corresponding to the question.

We introduce a knowledge-based type-aware text-to-SQL generator, TYPESQL. Based on the prior state-of-the-art SQLNet (Xu et al., 2017), TYPESQL employs a sketch-based approach and views the task as a slot filling problem (Figure 2). By grouping different slots in a reasonable way and capturing relationships between attributes, TYPESQL outperforms SQLNet by about 3.5% in half of the original training time.

Furthermore, natural language questions often contain rare entities and numbers specific to the underlying database. Some previous work (Agrawal and Srikant, 2003) already shows those words are crucial to many downstream tasks, such as infering column names and condition values in the SQL query. However, most of such key words lack accurate embeddings in popular pre-trained word embedding models. In order to solve this problem, TYPESQL assigns each word a type as an entity from either the knowledge graph, a column or a number. For example, for the question in Figure 1, we label "mort drucker" as PERSON according to our knowledge graph; "spoofed title," "artist" and "issue" as COLUMN since they are column names; and "88.5" as FLOAT. Incorporating this type information, TYPESQL further improves the state-of-the-art performance by about another 2% on the WikiSQL dataset, resulting in a final 5.5% improvement in total.

Moreover, most previous work assumes that user queries contain exact column names and entries. However, it is unrealistic that users always formulate their questions with exact column names and string entries in the table. To tackle this issue, when scaleability and privacy are not of a concern, the system needs to search databases to better understand what the user is querying. Our content-sensitive model TYPESQL + TC gains roughly 9% improvement compared to the content-insensitive model, and outperforms the previous content-sensitive model by 17.5%.

2 Related Work

Semantic parsing maps natural language to meaningful executable programs. The programs could

Proceedings of NAACL-HLT 2018, pages 588–594
New Orleans, Louisiana, June 1 - 6, 2018.

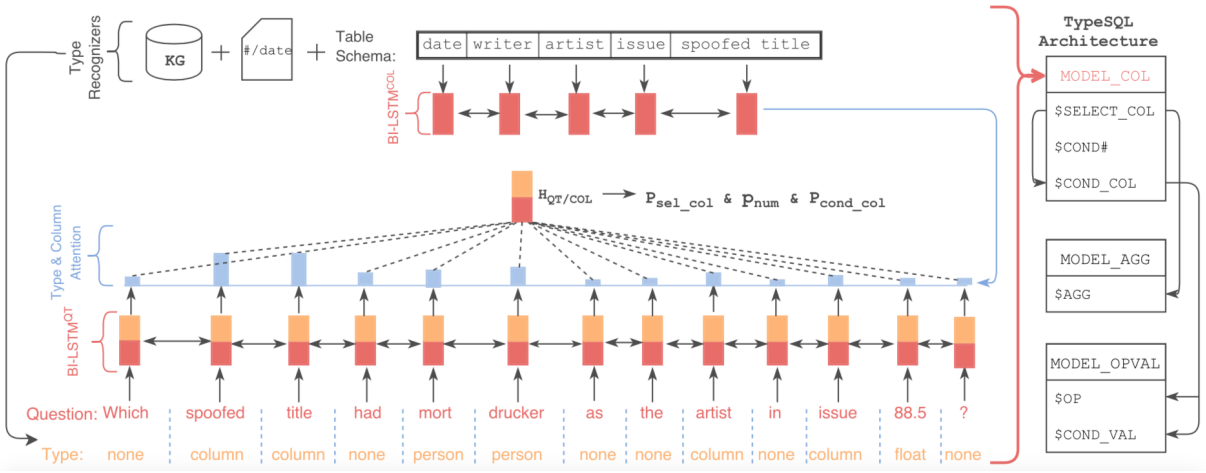

Figure 1: TYPESQL consists of three slot-filling models on the right. We only show MODEL_COL on the left for brevity. MODEL_AGG and MODEL_OPVAL have the similar pipelines.

be a range of representations such as logic forms (Zelle and Mooney, 1996; Zettlemoyer and Collins, 2005; Wong and Mooney, 2007; Das et al., 2010; Liang et al., 2011; Banarescu et al., 2013; Artzi and Zettlemoyer, 2013; Reddy et al., 2014; Berant and Liang, 2014; Pasupat and Liang, 2015). Another area close to our task is code generation. This task parses natural language descriptions into a more general-purpose programming language such as Python (Allamanis et al., 2015; Ling et al., 2016; Rabinovich et al., 2017; Yin and Neubig, 2017).

As a sub-task of semantic parsing, the text-to-SQL problem has been studied for decades (Warren and Pereira, 1982; Popescu et al., 2003, 2004; Li et al., 2006; Giordani and Moschitti, 2012; Wang et al., 2017b). The methods of the Database community (Li and Jagadish, 2014; Yaghmazadeh et al., 2017) involve more hand feature engineering and user interactions with the systems. In this work, we focus on recent neural network based approaches (Yin et al., 2016; Zhong et al., 2017; Xu et al., 2017; Wang et al., 2017a; Iyer et al., 2017). Dong and Lapata (2016) introduce a sequence-to-sequence approach to converting text to logical forms. Most of previous work focus on specific table schemas, which means they use a single database in both train and test. Thus, they don't generalize to new databases. Zhong et al. (2017) publish the WikiSQL dataset and propose a sequence-to-sequence model with reinforcement learning to generate SQL queries. In the problem definition of the WikiSQL task, the databases in the test set do not appear in the train and development sets. Also, the task needs to take different table schemas into account. Xu et al. (2017) further improve the results by using a SQL sketch based approach employing a sequence-to-set model.

SELECT $AGG $SELECT_COL
WHERE $COND_COL $OP $COND_VAL
(**AND** $COND_COL $OP $COND_VAL)*

Figure 2: SQL Sketch. The tokens starting with "$" are slots to fill. "*" indicates zero or more **AND** clauses.

3 Methodology

Like SQLNet, we employ a sketch-based approach and format the task as a slot filling problem. Figure 2 shows the SQL sketch. Our model needs to predict all slots that begin with $ in Figure 2.

Figure 1 illustrates the architecture of TYPESQL on the right and a detailed overview of one of three main models MODEL_COL on the left. We first preprocess question inputs by type recognition (Section 3.1). Then we use two bi-directional LSTMs to encode words in the question with their types and the column names separately (Section 3.2). The output hidden states of LSTMs are then used to predict the values for the slots in the SQL sketch (Section 3.3).

3.1 Type Recognition for Input Preprocessing

In order to create one-to-one type input for each question, we, first, tokenize each question into n-grams of length 2 to 6, and use them to search over the table schema and label any column name appears in the question as COLUMN. Then, we assign numbers and dates in the question into four self-

explanatory categories: INTEGER, FLOAT, DATE, and YEAR. To identify named entities, we search for five types of entities: PERSON, PLACE, COUNTRY, ORGANIZATION, and SPORT, on Freebase[1] using grams as keyword queries. The five categories cover a majority of entities in the dataset. Thus, we do not use other entity types provided by Freebase. Domain-specific knowledge graphs can be used for other applications.

In the case where the content of databases is available, we match words in the question with both the table schema and the content and labels of the columns as COLUMN and match the entry values as the corresponding column names. For example, the type in the Figure 1 would be [none, column, column, none, artist, artist, none, none, column, none, column, issue, none] in this case. Other parts in the Figure 1 keep the same as the content-insensitive approach.

3.2 Input Encoder

As shown in the Figure 1, our input encoder consists of two bi-directional LSTMs, BI-LSTM$^{\text{QT}}$ and BI-LSTM$^{\text{COL}}$. To encode word and type pairs of the question, we concatenate embeddings of words and their corresponding types and input them to BI-LSTM$^{\text{QT}}$. Then the output hidden states are $\mathbf{H}_{\text{QT}}$ and $\mathbf{H}_{\text{COL}}$, respectively.

For encoding column names, SQLNet runs a bi-directional LSTM over each column name. We first average the embeddings of words in the column name. Then, we run a single BI-LSTM$^{\text{COL}}$ between column names. This encoding method improves the result by 1.5% and cuts the training time by half. Even though the order of column names does not matter, we attribute this improvement to the fact that the LSTM can capture their occurrences and relationships.

3.3 Slot-Filling Model

Next, we predict values for the slots in the SQL sketch. For the slots in Figure 2, SQLNet has a separate model for each of them which do not share their trainable parameters. This creates five models for the five slots and one model for `$COND#` (12 BI-LSTMs in total). However, since the predict procedures of `$SELECT_COL`, `$COND_COL`, and `$COND#` are similar, we combine them into a single model. Additionally, `$COND_COL` depends on the output of `$SELECT_COL`, which reduces errors of predicting the same column in these two slots `$COND_COL` Moreover, we group `$OP` and `$COND_VAL` together because both depend on the outputs of `$COND_COL`. Furthermore, we use one model for `$AGG` because we notice that the `$AGG` model converges much faster and suffers from overfitting when combined with other models. Finally, TYPESQL consists of three models (Figure 1 right):

- MODEL_COL for `$SELECT_COL`, `$COND#` and `$COND_COL`
- MODEL_AGG for `$AGG`
- MODEL_OPVAL for `$OP` and `$COND_VAL`

where the parameters of BI-LSTM$^{\text{QT}}$ and BI-LSTM$^{\text{COL}}$ are shared in each model (6 BI-LSTMs in total).

Since all three models use the same way to compute the weighted question and type representation $\mathbf{H}_{\text{QT/COL}}$ using the column attention mechanism proposed in SQLNet, we first introduce the following step in all three models:

$$\alpha_{\text{QT/COL}} = \textbf{softmax}(\mathbf{H}_{\text{COL}}\mathbf{W}_{ct}\mathbf{H}_{\text{QT}}^{\top})$$
$$\mathbf{H}_{\text{QT/COL}} = \alpha_{\text{QT/COL}}\mathbf{H}_{\text{QT}}$$

where **softmax** applies the softmax operator over each row of the input matrix, $\alpha_{\text{QT/COL}}$ is a matrix of attention scores, and $\mathbf{H}_{\text{QT/COL}}$ is the weighted question and type representation. In our equations, we use $\mathbf{W}$ and $\mathbf{V}$ to represent all trainable parameter matrices and vectors, respectively.

MODEL_COL-$SELECT_COL $\mathbf{H}_{\text{QT/COL}}$ is used to predict the column name in the `$SELECT_COL`:

$$s = \mathbf{V}^{sel}\textbf{tanh}(\mathbf{W}_{c}^{sel}\mathbf{H}_{\text{COL}}^{\top} + \mathbf{W}_{qt}^{sel}\mathbf{H}_{\text{QT/COL}}^{\top})$$
$$P_{sel\_col} = \textbf{softmax}(s)$$

MODEL_COL-$COND# Unlike SQLNet, we compute number of conditions in the WHERE in a simpler way:

$$P_{num} = \textbf{softmax}\left(\mathbf{V}^{num}\textbf{tanh}(\mathbf{W}_{qt}^{num}\textstyle\sum_i \mathbf{H}_{\text{QT/COL}_i}^{\top})\right)$$

We set the maximum number of conditions to 4.

[1] https://developers.google.com/freebase/

	Dev			Test		
	Acc_{lf}	Acc_{qm}	Acc_{ex}	Acc_{lf}	Acc_{qm}	Acc_{ex}
Content Insensitive						
Dong and Lapata (2016)	23.3%	-	37.0%	23.4%	-	35.9%
Augmented Pointer Network (Zhong et al., 2017)	44.1%	-	53.8%	42.8%	-	52.8%
Seq2SQL (Zhong et al., 2017)	49.5%	-	60.8%	48.3%	-	59.4%
SQLNet (Xu et al., 2017)	-	63.2%	69.8%	-	61.3%	68.0%
TypeSQL w/o type-awareness (ours)	-	66.5%	72.8%	-	64.9%	71.7%
TypeSQL (ours)	-	**68.0%**	**74.5%**	-	**66.7%**	**73.5%**
Content Sensitive						
Wang et al. (2017a)	59.6%	-	65.2%	59.5%	-	65.1%
TypeSQL+TC (ours)	-	**79.2%**	**85.5%**	-	**75.4%**	**82.6%**

Table 1: Overall results on WikiSQL. Acc_{lf}, Acc_{qm}, and Acc_{ex} denote the accuracies of exact string, canonical representation, and execute result matches between the synthesized SQL with the ground truth respectively. The top six results are content-insensitive, which means only the question and table schema are used as inputs. The bottom two are content-sensitive, where the models use the question, the table schema, and the content of databases.

	Dev			Test		
	Acc_{agg}	Acc_{sel}	$\text{Acc}_{\text{where}}$	Acc_{agg}	Acc_{sel}	$\text{Acc}_{\text{where}}$
Seq2SQL (Zhong et al., 2017)	90.0%	89.6%	62.1%	90.1%	88.9%	60.2%
SQLNet (Xu et al., 2017)	90.1%	91.5%	74.1%	90.3%	90.9%	71.9%
TypeSQL (ours)	90.3%	**93.1%**	**78.5%**	90.5%	**92.2%**	**77.8%**
TypeSQL+TC (ours)	90.3%	**93.5%**	**92.8%**	90.5%	**92.1%**	**87.9%**

Table 2: Breakdown results on WikiSQL. Acc_{agg}, Acc_{sel}, and $\text{Acc}_{\text{where}}$ are the accuracies of canonical representation matches on AGGREGATOR, SELECT COLUMN, and WHERE clauses between the synthesized SQL and the ground truth respectively.

MODEL_COL-$COND_COL We find that SQLNet often selects the same column name in the $COND_COL as $SELECT_COL, which is incorrect in most cases. To avoid this problem, we pass the weighted sum of question and type hidden states conditioned on the column chosen in $SELECT_COL $\mathbf{H}_{\text{QT/SCOL}}$ (expended as the same shape of $\mathbf{H}_{\text{QT/COL}}$) to the prediction:

$$c = \mathbf{V}^{col}\textbf{tanh}(\mathbf{W}_c^{col}\mathbf{H}_{\text{COL}}^{\top} + \mathbf{W}_{qt}^{col}\mathbf{H}_{\text{QT/COL}}^{\top} + \mathbf{W}_{qt}^{scol}\mathbf{H}_{\text{QT/SCOL}}^{\top})$$
$$P_{cond\_col} = \textbf{softmax}(c)$$

MODEL_AGG-$AGG Given the weighted sum of question and type hidden states conditioned on the column chosen in $SELECT_COL $\mathbf{H}_{\text{QT/SCOL}}$, $AGG is chosen from {NULL, MAX, MIN, COUNT, SUM, AVG} in the same way as SQLNet:

$$P_{agg} = \textbf{softmax}\left(\mathbf{V}^{agg}\textbf{tanh}(\mathbf{W}_{qt}^{agg}\mathbf{H}_{\text{QT/SCOL}}^{\top})\right)$$

MODEL_OPVAL-$OP For each predicted condition column, we choose a $OP from $\{=, >, <\}$ by:

$$P_{op} = \textbf{softmax}\left(\mathbf{W}_t^{op}\textbf{tanh}(\mathbf{W}_c^{op}\mathbf{H}_{\text{COL}}^{\top} + \mathbf{W}_{qt}^{op}\mathbf{H}_{\text{QT/COL}}^{\top})\right)$$

MODEL_OPVAL-$COND_VAL Then, we need to generate a substring from the question for each predicted column. As in SQLNet, a bi-directional LSTM is used for the encoder. It employs a pointer network (Vinyals et al., 2015) to compute the distribution of the next token in the decoder. In particular, the probability of selecting the i-th token w_i in the natural language question as the next token in the substring is computed as:

$$v = \mathbf{V}_t^{val}\textbf{tanh}(\mathbf{W}_{qt}^{val}\mathbf{H}_{\text{QT}}^{i} + \mathbf{W}_c^{val}\mathbf{H}_{\text{COL}} + \mathbf{W}_h^{val}\mathbf{h})$$
$$P_{cond\_val} = \textbf{softmax}(v)$$

where $\mathbf{h}$ is the hidden state of the previously generated token. The generation process continues until the ⟨END⟩ token is the most probable next token of the substring.

4 Experiments

Dataset We use the WikiSQL dataset (Zhong et al., 2017), a collection of 87,673 examples of questions, queries, and database tables built from 26,521 tables. It provides train/dev/test splits such that each table is only in one split. This requires model to generalize to not only new questions but new table schemas as well.

Implementation Details We implement our model based on SQLNet (Xu et al., 2017) in Py-

Torch (Paszke et al., 2017). We concatenate pre-trained Glove (Pennington et al., 2014) and paraphrase (Wieting and Gimpel, 2017) embeddings. The dimensions and dropout rates of all hidden layers are set to 120 and 0.3 respectively. We use Adam (Kingma and Ba, 2015) with the default hyperparameters for optimization. The batch size is set to 64. The same loss functions in (Xu et al., 2017) are used. Our code is available at https://github.com/taoyds/typesql.

Results and Discussion Table 1 shows the main results on the WikiSQL task. We compare our work with previous results using the three evaluation metrics used in (Xu et al., 2017). Table 2 provides the breakdown results on AGGREGATION, SELECTION, and WHERE clauses.

Without looking at the content of databases, our model outperforms the previous best work by 5.5% on execute accuracy. According to Table 2, TYPESQL improves the accuracy of SELECT by 1.3% and WHERE clause by 5.9%. By encoding column names and grouping model components in a simpler but reasonable way, TYPESQL achieves a much higher result on the most challenging subtask WHERE clause. Also, the further improvement of integrating word types shows that TYPESQL could encode the rare entities and numbers in a better way.

Also, if complete access to the database is allowed, TYPESQL can achieve 82.6% on execute accuracy, and improves the performance of the previous content-aware system by 17.5%. Although (Zhong et al., 2017) enforced some limitations when creating the WikiSQL dataset, there are still many questions that do not have any column name and entity indicator. This makes generating the right SQLs without searching the database content in such cases impossible. This is not a critical problem for WikiSQL but is so for most real-world tasks.

5 Conclusion and Future Work

We propose TYPESQL for text-to-SQL which views the problem as a slot filling task and uses type information to better understand rare entities and numbers in the input. TYPESQL can use the database content to better understand the user query if it is not well-formed. TYPESQL significantly improves upon the previous state-of-the-art on the WikiSQL dataset.

Although, unlike most of the previous work, the WikiSQL task requires model to generalize to new databases, the dataset does not cover some important SQL operators such as JOIN and GROUP BY. This limits the generalization of the task to other SQL components. In the future, we plan to advance this work by exploring other more complex datasets under the database-split setting. In this way, we can study the performance of a generalized model on a more realistic text-to-SQL task which includes many complex SQL and different databases.

Acknowledgement

We thank Alexander Fabbri for his help in reviewing.

References

Rakesh Agrawal and Ramakrishnan Srikant. 2003. Searching with numbers. *IEEE Trans. Knowl. Data Eng.*, 15(4):855–870.

Miltiadis Allamanis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *ICML*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 2123–2132. JMLR.org.

Yoav Artzi and Luke Zettlemoyer. 2013. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association forComputational Linguistics*.

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*.

Jonathan Berant and Percy Liang. 2014. Semantic parsing via paraphrasing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1415–1425, Baltimore, Maryland. Association for Computational Linguistics.

Dipanjan Das, Nathan Schneider, Desai Chen, and Noah A. Smith. 2010. Probabilistic frame-semantic parsing. In *NAACL*.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*.

Alessandra Giordani and Alessandro Moschitti. 2012. Translating questions to sql queries with generative parsers discriminatively reranked. In *COLING (Posters)*, pages 401–410.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. *CoRR*, abs/1704.08760.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. *The 3rd International Conference for Learning Representations, San Diego*.

Fei Li and HV Jagadish. 2014. Constructing an interactive natural language interface for relational databases. *VLDB*.

Yunyao Li, Huahai Yang, and HV Jagadish. 2006. Constructing a generic natural language interface for an xml database. In *EDBT*, volume 3896, pages 737–754. Springer.

P. Liang, M. I. Jordan, and D. Klein. 2011. Learning dependency-based compositional semantics. In *Association for Computational Linguistics (ACL)*, pages 590–599.

Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Fumin Wang, and Andrew Senior. 2016. Latent predictor networks for code generation. In *ACL (1)*. The Association for Computer Linguistics.

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, pages 1470–1480.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. *NIPS 2017 Workshop*.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *EMNLP*, pages 1532–1543. ACL.

Ana-Maria Popescu, Alex Armanasu, Oren Etzioni, David Ko, and Alexander Yates. 2004. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *Proceedings of the 20th international conference on Computational Linguistics*, page 141. Association for Computational Linguistics.

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*, pages 149–157. ACM.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *ACL (1)*, pages 1139–1149. Association for Computational Linguistics.

Siva Reddy, Mirella Lapata, and Mark Steedman. 2014. Large-scale semantic parsing without question-answer pairs. *Transactions of the Association for Computational Linguistics*, 2:377–392.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2692–2700. Curran Associates, Inc.

Chenglong Wang, Marc Brockschmidt, and Rishabh Singh. 2017a. Pointing out sql queries from text. *Technical Report*.

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017b. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 452–466. ACM.

David HD Warren and Fernando CN Pereira. 1982. An efficient easily adaptable system for interpreting natural language queries. *Computational Linguistics*, 8(3-4):110–122.

John Wieting and Kevin Gimpel. 2017. Pushing the limits of paraphrastic sentence embeddings with millions of machine translations. *arXiv preprint arXiv:1711.05732*.

Yuk Wah Wong and Raymond J. Mooney. 2007. Learning synchronous grammars for semantic parsing with lambda calculus. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL-2007)*, Prague, Czech Republic.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436*.

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. Sqlizer: Query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA):63:1–63:26.

Pengcheng Yin, Zhengdong Lu, Hang Li, and Ben Kao. 2016. Neural enquirer: Learning to query tables in natural language. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 2308–2314.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *ACL (1)*, pages 440–450. Association for Computational Linguistics.

John M. Zelle and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *AAAI/IAAI*, pages 1050–1055, Portland, OR. AAAI Press/MIT Press.

Luke S. Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *UAI*.

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