SPARQLing Database Queries from Intermediate Question Decompositions

Irina Saparina and **Anton Osokin**
HSE University / Yandex / Moscow, Russia

Abstract

To translate natural language questions into executable database queries, most approaches rely on a fully annotated training set. Annotating a large dataset with queries is difficult as it requires query-language expertise. We reduce this burden using grounded in databases intermediate question representations. These representations are simpler to collect and were originally crowdsourced within the Break dataset (Wolfson et al., 2020). Our pipeline consists of two parts: a neural semantic parser that converts natural language questions into the intermediate representations and a non-trainable transpiler to the SPARQL query language (a standard language for accessing knowledge graphs and semantic web). We chose SPARQL because its queries are structurally closer to our intermediate representations (compared to SQL). We observe that the execution accuracy of queries constructed by our model on the challenging Spider dataset is comparable with the state-of-the-art text-to-SQL methods trained with annotated SQL queries. Our code and data are publicly available.[1]

1 Introduction

The difficulty of collecting and annotating datasets for the task of translating a natural language question to an executable database query is a significant obstacle to the progress of the technology. The most popular multi-database text-to-SQL dataset, Spider (Yu et al., 2018), has 10K questions, which is smaller compared to question answering datasets of other types: the DROP dataset with text paragraphs has 97K questions (Dua et al., 2019) and the GQA dataset with images has 22M questions (Hudson and Manning, 2019). The Spider dataset was created by 11 Yale students proficient in SQL, and it is difficult to scale such a process up.

Recently, Wolfson et al. (2020) proposed the Question Decomposition Meaning Representation, QDMR, which is a way to decompose a question into a list of "atomic" steps representing an algorithm for answering the question. Importantly, they developed a crowdsourcing pipeline to annotate QDMRs and showed that it can be used at scale: they collected 83K QDMRs for questions (all in English) coming from different datasets (including Spider) and released them in the Break dataset.

QDMRs resemble database queries but are not connected to any execution engine and cannot be run directly. Moreover, QDMRs were collected when looking only at questions and thus have no information about the database structure. Entities mentioned in QDMR steps usually have counterparts in the corresponding database but do not have links to them (grounding).

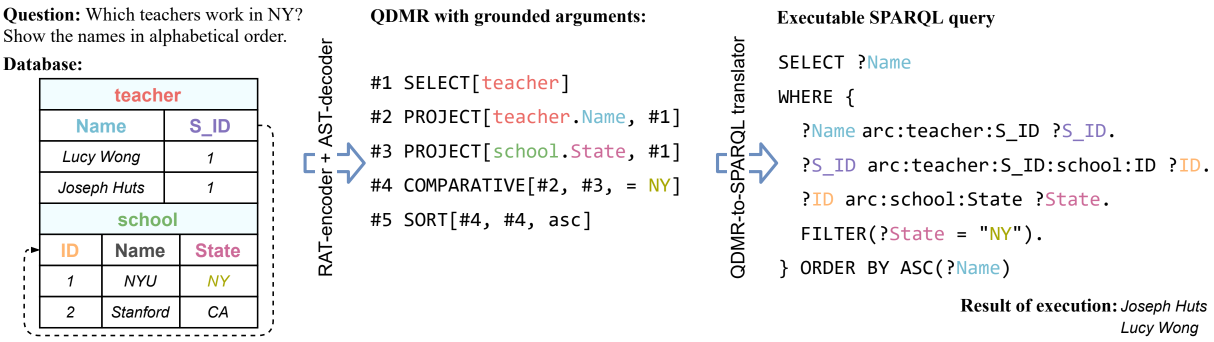
In this paper, we build a system for translating a natural language question first into QDMR and then into an executable query. We use modified QDMRs, where the entities described with text are replaced with their database groundings. Our system consists of two translators: a neural network for text-to-QDMR and a non-trainable QDMR-to-SPARQL transpiler. See Figure 1, for an illustration of our system.

In the text-to-QDMR part, we use an encoder-decoder model. Our encoder is inspired by RAT-transformer (Wang et al., 2020) and uses BERT (Devlin et al., 2019) or GraPPa (Yu et al., 2021). Our decoder is a syntax-guided network (Yin and Neubig, 2017) designed for our version of the QDMR grammar. We trained this model with full supervision, for which we automatically grounded QDMRs for a subset of Spider questions.

In the second part of the system, our goal was to translate grounded QDMRs into one of the existing query languages to benefit from the efficiency of database software. The most natural choice would be to use SQL, but designing such a translator

[1] https://github.com/yandex-research/sparqling-queries

Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pages 8984–8998
November 7–11, 2021. ©2021 Association for Computational Linguistics



Question: Which teachers work in NY? Show the names in alphabetical order.

Database:

teacher	
Name	S_ID
Lucy Wong	1
Joseph Huts	1

school		
ID	Name	State
1	NYU	NY
2	Stanford	CA

QDMR with grounded arguments:

```
#1 SELECT[teacher]
#2 PROJECT[teacher.Name, #1]
#3 PROJECT[school.State, #1]
#4 COMPARATIVE[#2, #3, = NY]
#5 SORT[#4, #4, asc]
```

Executable SPARQL query

```
SELECT ?Name
WHERE {
  ?Name arc:teacher:S_ID ?S_ID.
  ?S_ID arc:teacher:S_ID:school:ID ?ID.
  ?ID arc:school:State ?State.
  FILTER(?State = "NY").
} ORDER BY ASC(?Name)
```

Result of execution: *Joseph Huts*
Lucy Wong

Figure 1: Overall map of our approach: we feed a question and a database schema into the encoder-decoder model to obtain the grounded QDMR. The grounded QDMR is then fed into our QDMR-to-SPARQL translator to obtain an executable SPARQL query. The generated query is executed on the database in the RDF format.

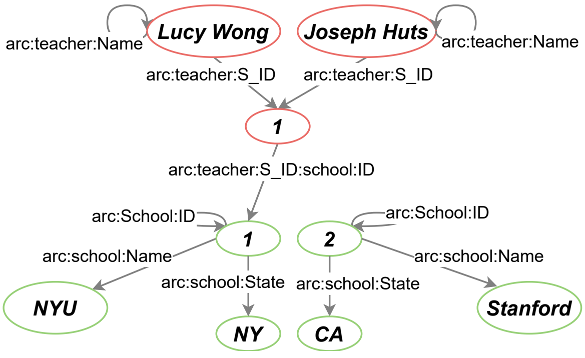


Figure 2: Database from Figure 1 converted to the RDF format (the RDF graph). The red nodes correspond to the values from **teacher** table, the green ones - to the values from **school** table. Arcs correspond to the relations between primary key and other values of the same row (`arc:tbl:col`) and along the foreign keys (`arc:t_src:c_src:t_tgt:c_tgt`).

was difficult due to structural differences between QDMR and SQL. Instead, we implement a translator from QDMR to SPARQL,[2] which is a query language for databases in the Resource Description Framework (RDF) format (Prud'hommeaux and Seaborne, 2008; Harris and Seaborne, 2013). SPARQL is a standard made by the World Wide Web Consortium and is recognized as one of the key technologies of the semantic web. See Figure 2 for an example of the RDF database.

We evaluated our system with the execution accuracy metric on the Spider dataset (splits by Wolfson et al., 2020) and compared it with two strong baselines: text-to-SQL systems BRIDGE (Lin et al., 2020) and SmBoP (Rubin and Berant, 2021) from the top of the Spider leaderboard. On the cleaned-up validation set, our system outperforms both baselines. On the test set with original annotation, our system is in-between the baselines. Additionally, we experimented with training our models on extra data: items from Break without schema but with QDMRs. This teaser experiment showed potential for further improvements.

This paper is organized as follows. Sections 2 and 3 present two main parts of our system. Section 4 contains the experimental setup, Section 5 – our results. We review related works in Section 6 and conclude in Section 7.

2 QDMR-to-SPARQL translator

2.1 QDMR logical forms

Question Decomposition Meaning Representation (QDMR) introduced by Wolfson et al. (2020) is an intermediate format between a question in a natural language (tested in English) and an executable query in some formal query language. QDMR is a sequence of steps, and each step corresponds to a separate logical unit of the question (see Table 1). A QDMR step can refer to one of the previous steps, allowing one to organize the steps into a graph.

We work with QDMR logical forms (LF), which can be automatically obtained from the text-based QMDRs, e.g., with the rule-based method of Wolfson et al. (2020). Steps of a logical form are derived from the corresponding steps of QDMR. Each step of LF includes an operator and its arguments. We show some operators in Table 2 and provide the full list in Appendix A.[3]

[2] SPARQL is a recursive acronym for SPARQL Protocol and RDF Query Language.

[3] Differently from Wolfson et al. (2020) we merged the operation `FILTER` into `COMPARATIVE` due to their similarity and excluded `ARITHMETIC`, `BOOLEAN` and undocumented `COMPARISON` because they are extremely rare in the Spider part of Break.

Question:	For each state, how many teachers are there?
QDMR (Break)	#1 return states #2 return teachers in #1 #3 return number of #2 for each #1 #4 return #1 and #3
QDMR logical form (Break)	#1 SELECT[states] #2 PROJECT[teachers in #REF, #1] #3 GROUP[count, #2, #1] #4 UNION[#1, #3]
grounded QDMR (ours)	#1 SELECT[School.State] #2 PROJECT[teacher, #1] #3 GROUP[count, #2, #1] #4 UNION[#1, #3]

Table 1: Examples of different QDMR formats: textual QDMR, QDMR logical form (from Break) and our version of QDMR with grounded arguments.

2.2 Grounding QDMRs in databases

QDMR logical forms are similar to the programmed queries but are not connected to any execution engine and cannot be executed directly. To execute these LFs using knowledge from a database, one needs to associate their arguments with the entities of the database: tables, columns, values. We refer to this association as grounding and provide the details below.

Arguments of LF operators can be of different types (see Table 2 and Appendix A) and some types require groundings. Type `ref` indicates a reference to one of the existing LF steps. Type `text` corresponds to a text argument that needs to be grounded to a table, column or value in the database. Type `choice` corresponds to the choice among a closed list of possible options, and type `bool` corresponds to the True/False choice.

There are also a few edge cases that require special processing. First, the `value` argument of the `COMPARATIVE` operator can be either `ref` or `text`. Second, the `operator` argument of `AGGREGATE`/`GROUP` can actually be grounded to a column. We introduced this exception because a database can contain only the aggregated information without information about individual instances. As the QDMR annotation is built without looking at the database it cannot distinguish the two cases. In the example of Table 1, if the database has a column `num_teachers` in the table `school` we would need to ground `count` to the column `num_teachers`.

We describe our procedure for annotating LF arguments with groundings in Section 4.2.

2.3 Executable queries in SPARQL

To convert a QDMR LF with grounded step arguments into an actually executable query, it is beneficial to translate QDMR into one of the existing query languages to use an existing efficient implementation at the test time. In this paper, we translate QDMR queries into SPARQL, a language for querying databases in the graph-based RDF format (Prud'hommeaux and Seaborne, 2008; Harris and Seaborne, 2013). Next, we briefly overview the RDF database format and SPARQL and then describe our algorithm for translating grounded LFs into SPARQL queries.

RDF format. In RDF, data is stored as a directed multi-graph, where the nodes correspond to the data elements and the arcs correspond to relations. RDF-graphs are usually defined by sets of subject-predicate-object triples, where each triple defines an arc: the subject is the source node, the predicate is the type of relation and the object is the target node.

Relational data to RDF. To evaluate our approach on the Spider dataset containing relational databases (in the SQLite format), we convert relational databases to the RDF format. The conversion is inspired by the specification of Arenas et al. (2012). For each table row of the relational database, we add to the RDF graph a set of triples corresponding to each column. For the primary key column[4] `key` of a table `tbl`, we create a triple with the self-link `arc:tbl:key` pointing from the key element to itself. For any other column `col` in the table `tbl`, we create a triple with the separate edge type `arc:tbl:col`, which connects the primary key element of a row to the corresponding element in `col`. For each foreign key of the database, we create an arc type `arc:t_src:c_src:t_tgt:c_tgt` (here the target column `c_tgt` has to be a key). Then we add to the RDF graph the triples with these foreign-key relations. See Figure 2 with an example of the RDF graph for the database of Figure 1.

SPARQL. In a nutshell, a SPARQL query is a set of triple patterns where some elements are replaced with variables. The execution happens by searching the RDF graph for subgraphs that match the patterns. For example, a query

```
SELECT ?State WHERE {
  ?ID arc:school:State ?State.}
```

[4] For simplicity, we assume that each table has a single-column primary key (otherwise, we add a new `ID` column).

Operator	Arguments:their types	Description
SELECT	[subj:text, distinct:bool]	Select subj (possibly distinct values)
PROJECT	[proj:text, subj:ref]	Select proj related to subj (possibly distinct values)
COMPARATIVE	[subj:ref, attr:ref, comp:choice, value:text/ref, distinct:bool]	Select subj such that related attr compares (using $=, \neq, >, <, \geq, \leq$, like) to value (possibly distinct values)
GROUP	[subj:ref, attr:ref, op:choice]	Group subj such that attr has same values (aggr. with op)

Table 2: QDMR operators and their arguments with types. See Appendix A for the full version –Table 8.

to the RDF graph of Figure 2 searches for pairs of nodes that are connected with arcs of type arc:school:State. Entries starting with symbol ? represent variables. See Figure 1 for an example of a more complicated query.

SPARQL also supports subqueries and aggregators, the GROUP, SORT, UNION, MINUS keywords, etc. See, e.g., the Wikidata SPARQL tutorial[5] for a detailed overview of SPARQL features.

Translating grounded QDMR to SPARQL. We implemented a translator from a grounded QDMR LF into SPARQL. Note that LFs do not have a formal specification defining the execution, so our translator fills in the formal meaning. Our translator recursively constructs graph patterns that contain a result of LF steps. When processing a step, the method first constructs one or several patterns for the step arguments and then connects them into another pattern. At the beginning of the process, we request the method to construct the pattern containing the last QDMR step, which corresponds to the query output. We provide the details of our translator in Appendix A.

3 Text-to-QDMR parser

In this section, we describe our approach to generating a grounded QDMR LF from a given question and a database schema. Our encoder consists of BERT-like pretrained embeddings (Devlin et al., 2019; Yu et al., 2021) and a relation-aware transformer (Wang et al., 2020). Our decoder is an LSTM model that generates an abstract syntax tree in the depth-first traversal order (Yin and Neubig, 2017).

3.1 Encoder

In our task, the input is a sequence of question tokens and a set of database entities eligible for grounding: tables, columns, and extracted values.

To choose values from a database, we use string matching between question tokens and database values (see Appendix B). Additionally, we extract numbers and dates from the question that can be valid comparative values not from the database. To avoid ambiguity of the encoding, we combine the multiple identical values from different columns into one.

Following Huang et al. (2018); Zhang et al. (2019); Wang et al. (2020), the input tokens of four types (question, table, column and value) are interleaved with [SEP], combined into a sequence and encoded: we experiment with BERT (Devlin et al., 2019) and GraPPa (Yu et al., 2021). The obtained representations are fed into the relation-aware transformer, RAT (Wang et al., 2020).

RAT module. RAT (Wang et al., 2020) is based on relation-aware self-attention layer (Shaw et al., 2018) . Unlike the standard self-attention in the transformer model (Vaswani et al., 2017), this layer explicitly adds embeddings $\boldsymbol{r}_{ij}$ that encode relations between two inputs $\boldsymbol{x}_i, \boldsymbol{x}_j$. The RAT self-attention weights are $\alpha_{ij} = \text{softmax}\Big(\frac{\boldsymbol{x}_i W_Q (\boldsymbol{x}_j W_K + \boldsymbol{r}_{ij})^\top}{\sqrt{d}}\Big)$, where W_K, W_Q, d are the standard self-attention parameters.

The relations between the columns and tables come from the schema structure, e.g., the table – primary key and foreign key relations. We also have relations based on matches: question – table and question – column matches based on the n-gram comparison (Guo et al., 2019) and question – value matches from our value extracting procedure.

3.2 Decoder

The decoder is a recurrent model with LSTM cells that generates an abstract syntax tree (AST) in the depth-first traversal order (Yin and Neubig, 2017). At each prediction, the decoder selects one of the allowed outputs, the list of allowed outputs is defined by our QDMR grammar (see Appendix C). The output can be the grammar rule (transition to

[5] https://www.wikidata.org/wiki/Wikidata:SPARQL_tutorial

a new node in AST), the grounding choice or the previous step number (leaf nodes in AST).

To predict grammar rules, we use the same modules as in the RAT-SQL model (Wang et al., 2020). The decoder predicts comparator, aggregator and sort directions using the output of MLP. For table, column or value grounding, we use the pointer network attention mechanism (Vinyals et al., 2015). To predict a reference to a previous QDMR step, we use an MLP with a mask in the output softmax. To avoid incorrect QDMR output, we use several restrictions in the decoding process. Most of them are in the prediction of comparative arguments, e.g., we check type consistency (see Appendix D).

3.3 Training

We follow the RAT-SQL (Wang et al., 2020) training procedure in the main aspects. We use the standard teacher forcing approach for autoregressive models. We found that an additional alignment loss proposed for RAT-SQL did not lead to any improvements in our case, so we trained the models with the cross-entropy loss with label smoothing. See Appendix B for implementation details.

Augmentations. We randomly permute tables, columns and values when training. We experimented with a random choice of QDMR graph linearization at training but did not observe performance improvements. We also tried to randomly select one of the multiple available QDMR groundings, but it did not help as well.

4 Experiment setup

4.1 Data

For training and evaluation, we use the part of the Break dataset that corresponds to the Spider dataset.[6] Data includes questions and databases from Spider, QDMR logical forms from Break and groundings that we collected. Automatic grounding annotation is challenging, but we are able to annotate with target groundings more than 60% of the Break data (see Section 4.2). Our splits are based on the Break splits but take into account the grounding annotation. The Break dataset does not include the Spider test, as it is hidden, while the Break dev and test are the halves of the Spider dev. The gold QDMR and grounding annotation on the Break test is also hidden. The overall dataset statistics are shown in Table 3.

Dataset	Train	Dev	Test
Original Spider	8695	1034	2147
- with Break	6921	502	521
- with groundings	4350	445	-

Table 3: Dataset statistics for the original Spider, part of Spider with QDMR annotations from Break and part of Spider with QDMRs and groundings. Break dev and test are splits of original Spider dev. Break test is hidden, so we do not have annotation for this part.

We fixed typos and annotation errors in some train and dev examples. We also corrected some databases on train and dev: we deleted trailing white spaces in values (led to mismatches between SQL query and database) and added missing foreign keys (necessary for our SPARQL generator) based on the procedure of Lin et al. (2020). We kept the test questions and SQL queries unchanged from the original Spider dataset, which implied that some dataset errors could degrade comparisons of SQL and SPARQL results.

4.2 Annotating Groundings for LFs

We process LFs from the Break dataset in several stages. At the first stage, we iterate over all the operators and make their arguments compatible with our specification (see Table 2).

At the second stage, we collect candidate groundings for each argument that requires grounding. At this stage, we use all available sources of information: text-based similarity between the text argument and the names of the database entities, the corresponding SQL query from Spider, explicit linking between the question tokens and the elements of the schema released by Lei et al. (2020). Importantly, we can match the output of LF to the output of the SQL query and propagate groundings inside LF, which allows to obtain many high-quality groundings. At the third stage, we use collected candidate groundings and group them in all possible ways to obtain candidate LFs with all arguments grounded. Then, for each candidate LF, we run our QDMR-to-SPARQL translator and execute the obtained query. We accept the candidate if there are no failures in the pipeline and the result of the SPARQL query equals the result of the SQL one. Finally,

[6] The Break dataset also contains QDMRs for other text-to-SQL datasets, e.g., single-database ATIS and GeoQuery. Comparison in the regime of fine-tuning on a specific database is also interesting, but baseline and our codebases failed due to the limitations of the SQL parsers (coming from Spider). This issue might be resolved by switching to a different SQL parser but it appeared technically infeasible at the time of writing.

we included the question in the dataset if we had accepted at least one grounded LF. Note that we can accept several versions of grounding for each question. We cannot figure out which one is better at this point, so we can either pick one randomly or use all of them at training.

4.3 Evaluation Metric

For evaluation on the Spider dataset, most text-to-SQL methods use the metric called exact set matching without values. This metric compares only some parts of SQL queries, e.g., values in conditions are not used, and sometimes incomplete non-executable queries can achieve high metric values. As our approach does not produce any SQL query at all, this metric is not applicable.

Instead, we use an execution-based metric, a.k.a. execution accuracy. This metric compares the results produced by the execution of queries (allowing arbitrary permutation on the output columns). Recently, the Spider leaderboard started supporting this metric, but submitting directly to the leaderboard is still not possible for us because the exposed interface requires SQL queries. We modify the Spider execution accuracy evaluation in such a way that it can support any query language that can be executed and provide results. When comparing the results of SPARQL to the results of SQL, we faced several challenges:

- the order of output columns in SQL does not match the order in the question;
- in Spider, when selecting relations w.r.t. argmin or argmax there is no consistent policy whether to pick all the rows satisfying the constraints or only one of them;
- the order of rows in the output of SQL is stable, but the order of rows in the output of SPARQL varies depending on minor launching conditions;
- in SPARQL, sorting is unstable (can arbitrarily change elements with equal sorting key values), but SQL sorting is stable;

The first two points can make SQL-to-SQL comparisons invalid as well, and the others affect only SQL-to-SPARQL comparisons.

To resolve these issues, we implemented the metric supporting SQL-to-SQL, SQL-to-SPARQL, SPARQL-to-SPARQL comparisons with the following properties:

- we reorder the columns of the outputs based on the columns the output values come from. If the matching fails, we try to compare output tables with the given order of columns;
- if one of the outputs is from an SQL query ending with “ORDER BY$\cdots$LIMIT 1”, we check that the produced one row is contained in another output;
- if one of the outputs has done unstable sorting, we allow it to provide a key w.r.t. which the sorting was done and try to match the order of the rows in another output by swapping the rows with identical sorting-key values;
- before comparison, we extract the column types from both outputs and convert each value to the standardized representation.

Model	Train	Pretrain	Dev	Test
BRIDGE	full	BERT	68.3	61.4
SmBoP	full	GraPPa	74.2	64.1
BRIDGE	subset	BERT	67.4	60.3
SmBoP	subset	GraPPa	71.7	**64.9**
Ours	subset	BERT	79.3	60.8
Ours	subset	GraPPa	**80.4**	62.0

Table 4: Execution accuracy of our model compared to state-of-the-art text-to-SQL methods on our development and test sets.

Train	Pretrain	Augs	Dev	Test
subset	BERT	+	79.3	60.8
full Break	BERT	-	78.4	61.8
full Break	BERT	+	78.9	61.8
subset	GraPPa	+	**80.4**	62.0
full Break	GraPPa	-	74.6	**62.6**
full Break	GraPPa	+	73.9	61.4

Table 5: Execution accuracy of our model trained on only the Spider subset of Break compared to using additional data from Break (on our development and test sets).

5 Results

5.1 Comparison with text-to-SQL methods

First, we compare our approach to state-of-the-art text-to-SQL methods (that generate full executable queries) BRIDGE (Lin et al., 2020) and SmBoP (Rubin and Berant, 2021), both from the top of the Spider leaderboard. See Table 4 for the results. As our training data includes only 50% of the original Spider train, we add to the comparison BRIDGE and SmBoP models trained on the same data subset. We use the official implementations of both models.

All models are trained together with finetuning pretrained contextualized representations: BRIDGE encoder uses BERT, SmBoP encoder uses GraPPa, our model has both BERT and GraPPa versions.

We choose the final model of each training run of our system based on the best dev result from the last 10 checkpoints with the step of 1000 iterations. For BRIDGE and SmBoP, we used the procedures provided in the official implementations (they similarly look at the same dev set). The estimated std of our model is 0.9 on the dev set (estimated via retraining our BERT-based model with 5 different random seeds).

On the development set, our models achieve better execution accuracy than text-to-SQL parsers even trained on full Spider data. On the test set, our models outperform BRIDGE but not SmBoP when trained on the same amount. See Table 6 for qualitative results of our GraPPa-based model.

We did not include the results of RAT-SQL (Wang et al., 2020) in Table 4, because this model was trained to optimize exact set matching without values, so the model output contains placeholders instead of values. The model trained on full Spider reproduces the exact matching scores shown by Wang et al. (2020) but gives only 40.2% execution accuracy on dev and 39.9% on test. Correct predictions mostly came from correct SQL queries without values. We also tried the available feature of value prediction in the official implementation of RAT-SQL and obtained better execution accuracy scores (48.5% on dev and 46.4% on test), but they were still very low.

5.2 Additional training data from Break

The Break dataset contains QDMR annotations for several question answering datasets, so we tried to enrich training on Spider with QDMRs from other parts of Break. Table 5 shows the execution accuracy on our dev and test in these settings. Adding training data for both versions of the model leads to performance improvement on the test set, but slightly decreases the dev set results.

When training with the data from other parts of Break, we simply assume that the schema is empty and use all the textual QDMR arguments as values. More careful exploration of additional QDMR data is left for future work.

5.3 Ablation study

Table 7 presents results of ablations on the development set. First, note that disabling augmentations

Q:	How many concerts are there in year 2014 or 2015?

SQL:	SELECT count(*) FROM concert
	WHERE YEAR = 2014 OR YEAR = 2015

Ours ✓	#1 SELECT[concert] #2 PROJECT[concert.Year, #1] #3 COMPARATIVE[#1,#2,=2014] #4 COMPARATIVE[#1,#2,=2015] #5 UNION[#3, #4] #6 AGGREGATE[count, #5]
Q:	Show location and name for all stadiums with a capacity between 5000 and 10000.

SQL:	SELECT location, name FROM stadium
	WHERE capacity
	BETWEEN 5000 AND 10000

Ours ✓	#1 SELECT[stadium] #2 PROJECT[stadium.Capacity, #1] #3 COMPARATIVE[#1,#2, ≥5000] #4 COMPARATIVE[#1,#2, ≤10000] #5 INTERSECTION[#1, #3, #4] #6 PROJECT[stadium.Location, #5] #7 PROJECT[stadium.Name, #5] #8 UNION[#6, #7]
Q:	What is the year that had the most concerts?

SQL:	SELECT year FROM concert
	GROUP BY year
	ORDER BY count(*) DESC LIMIT 1

Ours ✓	#1 SELECT[concert.Year] #2 PROJECT[concert, #1] #3 GROUP[count, #2, #1] #4 SUPERLATIVE[max, #1, #3]
Q:	What are the names of the stadiums without any concerts?

SQL:	SELECT name FROM stadium
	WHERE stadium_id NOT IN
	(SELECT stadium_id FROM concert)

Ours ✓	#1 SELECT[stadium] #2 COMPARATIVE[#1, #1, concert] #3 DISCARD[#1, #2] #4 PROJECT[stadium.Name, #3]
Q:	What are the number of concerts that occurred in the stadium with the largest capacity?

SQL:	SELECT count(*) FROM concert
	WHERE stadium_id =(
	SELECT stadium_id FROM stadium
	ORDER BY capacity DESC LIMIT 1)

Ours ✓	#1 SELECT[stadium] #2 PROJECT[stadium.Capacity, #1] #3 SUPERLATIVE[max, #1, #2] #4 PROJECT[concert, #3] #5 AGGREGATE[count, #4]
Q:	What is the average and maximum capacities for all stadiums?
SQL:	SELECT avg(capacity), max(capacity) FROM stadium
Ours ✗	#1 SELECT[stadium] #2 PROJECT[stadium.Average, #1] #3 AGGREGATE[avg, #2] #4 AGGREGATE[max, #2] #5 UNION[#3, #4]

Table 6: Qualitative results of our GraPPa-based model. ✓and ✗ denote correct and incorrect execution results respectively.

Model	Pretrain	Dev
Base	BERT	79.3
- w/o augmentations	BERT	75.7
- w/o schema relations	BERT	68.1
- with default relations	BERT	65.4
- w/o relation-aware layers	BERT	51.0
Base	GraPPa	80.4
- w/o augmentations	GraPPa	75.7

Table 7: Execution accuracy for our ablation study.

in both models decreases the execution accuracy.

Next, we tested different configurations of RAT-encoder:

- without relations that come from the schema structure (e.g., the table – primary key and foreign key relations);
- with the small number of default relations: without distinguishing table, column or value, because these elements are considered as elements of one unified grounding type;
- the regular transformer instead of RAT.

The model without schema relations lost 11% on dev, which shows that encoding schema with RAT-encoder is an important part of the model. This also limits the use of additional data from Break, where schemas do not exist. The variety of relations in RAT-encoder is also important, as RAT itself. Our findings are consistent with the ablations of Wang et al. (2020).

6 Related Work

Text-to-SQL. The community has recently made significant progress and moved from fixed-schema datasets like ATIS or GeoQuery (Popescu et al., 2003; Iyer et al., 2017) to the WikiSQL or Overnight datasets with multiple single-table schemas (Wang et al., 2015; Zhong et al., 2017) and then to the Spider dataset with multiple multi-table multi-domain schemas (Yu et al., 2018). Since the release of Spider, the accuracy has moved up from around 10% to 70%.

Most recent systems are structured as encoder-decoder networks. Encoders typically consist of a module fine-tuned from a pretrained language model like BERT (Devlin et al., 2019) and a module for incorporating the schema structure. Guo et al. (2019); Zhong et al. (2020); Lin et al. (2020) represented schemas as token sequences, Bogin et al. (2019a,b) used graph neural networks and Wang et al. (2020) used relation-aware transformer, RAT, to encode a graph constructed from an input schema. In this paper, we use the RAT module to encode the schema but enlarge the encoded graph by adding value candidates as nodes.

Decoders are typically based on a grammar representing a subset of SQL and produce output tokens in the depth-first traversal order of an abstract syntax tree, AST, following Yin and Neubig (2017). A popular choice for such a grammar is to use SemQL of Guo et al. (2019) or to use a lighter grammar with more intensive consistency checks inside beam search like in BRIDGE (Lin et al., 2020). Recently, Rubin and Berant (2021) proposed a different approach to decoding based on bottom-up generating of sub-trees on top of the relational algebra of SQL. In our paper, we follow the standard AST-based approach but for the grammar describing grounded QDMRs. We also use some consistency checks and the decoding time to prevent some easily avoidable inconsistencies.

There is also a line of work on weakly-supervised learning of text-to-SQL semantic parsers, where SQL queries or logical forms for the training set are not available at all. Some works (Min et al., 2019; Wang et al., 2019; Agarwal et al., 2019; Liang et al., 2018) reported results on the WikiSQL dataset, Wang et al. (2021) worked on GeoQuery and Overnight datasets. We are not aware of any works reporting weakly-supervised results on the multi-table Spider dataset.

Pretraining on text and tables. One possible direction inspired by the success of pretraining language models on large text corpora is to pretrain model on data with semantically connected text and tables. Yin et al. (2020, TaBERT) and Herzig et al. (2020, TaPas) used text-table pairs extracted from sources like Wikipedia for pretraining. Yu et al. (2021, GraPPa) used synthetic question-SQL pairs. Deng et al. (2021, STRUG) used the table-to-text dataset of Parikh et al. (2020, ToTTo). Shi et al. (2021, GAP) used synthetic data generated by the models for SQL-to-text and table-to-text auxiliary tasks. In this paper, we do not pretrain such models but experiment with GraPPa as the input encoder.

QDMR. Together with the Break dataset, Wolfson et al. (2020) created a task of predicting QDMRs given questions in English. As a baseline, they created a seq2seq model enhanced with a copy mechanism of Gu et al. (2016). Recently, Hasson and Berant (2021) built a QDMR parser that is based on dependency graphs and uses RAT modules. Differently from this line of work, we use a modified version of QDMRs, and our models never actually predict QDMR arguments as text but always directly their groundings.

SPARQL. SPARQL was used in several lines of work on semantic parsing for querying knowledge bases. The SEMPRE system of Berant et al. (2013) relied on SPARQL to execute logical forms on the Freebase knowledge base. Yih et al. (2016) and Talmor and Berant (2018) created the WebQuestions and ComplexWebQuestions datasets, respecively, where annotations were provided in the form of SPARQL queries. A series of challenges on Question Answering over Linked Data Challenge, QALD (Lopez et al., 2013), and the LC-QuAD datasets (Trivedi et al., 2017; Dubey et al., 2019) targeted the generation of SPARQL queries directly. Our paper is different from these lines of work as we rely on supervision via QDMRs and not SPARQL directly.

There also exist several lines of works on converting queries from/to SPARQL, and the problems are difficult. See, e.g., the works of Michel et al. (2019); Abatal et al. (2019) and references therein.

7 Conclusion

In this paper, we proposed a way to use the recent QDMR format (Wolfson et al., 2020) as annotation for generating executable queries to databases given a question in a natural language. Using QDMRs is beneficial because they can be collected through crowdsourcing potentially easier than correct database queries. Our system consists of two main parts. First, we have a learned text-to-QDMR translator that we built on top of the recent RAT-SQL system (Wang et al., 2020) and trained on an annotated with QDMRs part of the Spider dataset. Second, we have a non-trainable QDMR-to-SPARQL translator, which generates queries executable on databases in the RDF format. We evaluated our system on the Spider dataset and showed it to perform on par with the modern text-to-SQL methods (BRIDGE and SmBoP) trained with full supervision in the form of SQL queries. We also showed that additional QDMR annotations for questions not aligned with any databases could further improve the performance. The improvement shows great potential for future work.

Acknowledgements

This research was supported in part through computational resources of HPC facilities at HSE University (Kostenetskiy et al., 2021).

References

Ahmed Abatal, Khadija Alaoui, Larbi Alaoui, and Mohamed Bahaj. 2019. SQL2SPARQL4RDF: Automatic SQL to SPARQL conversion for RDF querying. In *Proceedings of the 4th International Conference on Big Data and Internet of Things*. Association for Computing Machinery.

Rishabh Agarwal, Chen Liang, Dale Schuurmans, and Mohammad Norouzi. 2019. Learning to generalize from sparse and underspecified rewards. In *Proceedings of the 36th International Conference on Machine Learning (ICML), Long Beach, California, PMLR 97*.

Marcelo Arenas, Alexandre Bertails, Eric Prud'hommeaux, and Juan Sequeda. 2012. A direct mapping of relational data to RDF. W3C Recommendation, World Wide Web Consortium.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on Freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544, Seattle, Washington, USA. Association for Computational Linguistics.

Ben Bogin, Jonathan Berant, and Matt Gardner. 2019a. Representing schema structure with graph neural networks for text-to-SQL parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4560–4565, Florence, Italy. Association for Computational Linguistics.

Ben Bogin, Matt Gardner, and Jonathan Berant. 2019b. Global reasoning over database structures for text-to-SQL parsing. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3659–3664, Hong Kong, China. Association for Computational Linguistics.

Xiang Deng, Ahmed Hassan Awadallah, Christopher Meek, Oleksandr Polozov, Huan Sun, and Matthew Richardson. 2021. Structure-grounded pretraining for text-to-SQL. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1337–1350, Online. Association for Computational Linguistics.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference* *of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. 2019. DROP: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2368–2378, Minneapolis, Minnesota. Association for Computational Linguistics.

Mohnish Dubey, Debayan Banerjee, Abdelrahman Abdelkawi, and Jens Lehmann. 2019. LC-QuAD 2.0: A large dataset for complex question answering over Wikidata and DBpedia. In *Proceedings of the 18th International Semantic Web Conference (ISWC)*.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. 2016. Incorporating copying mechanism in sequence-to-sequence learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1631–1640, Berlin, Germany. Association for Computational Linguistics.

Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards complex text-to-SQL in cross-domain database with intermediate representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4524–4535, Florence, Italy. Association for Computational Linguistics.

Steve Harris and Andy Seaborne. 2013. SPARQL 1.1 query language. W3C Recommendation, World Wide Web Consortium.

Matan Hasson and Jonathan Berant. 2021. Question decomposition with dependency graphs. In *Automated Knowledge Base Construction (AKBC)*.

Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Eisenschlos. 2020. TaPas: Weakly supervised table parsing via pre-training. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4320–4333, Online. Association for Computational Linguistics.

Po-Sen Huang, Chenglong Wang, Rishabh Singh, Wen-tau Yih, and Xiaodong He. 2018. Natural language to structured query generation via meta-learning. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 732–738, New Orleans, Louisiana. Association for Computational Linguistics.

Drew A. Hudson and Christopher D. Manning. 2019. GQA: A new dataset for real-world visual reasoning and compositional question answering. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 963–973, Vancouver, Canada. Association for Computational Linguistics.

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*.

P. S. Kostenetskiy, R. A. Chulkevich, and V. I. Kozyrev. 2021. HPC resources of the higher school of economics. *Journal of Physics: Conference Series*, 1740.

Wenqiang Lei, Weixin Wang, Zhixin Ma, Tian Gan, Wei Lu, Min-Yen Kan, and Tat-Seng Chua. 2020. Re-examining the role of schema linking in text-to-SQL. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6943–6954, Online. Association for Computational Linguistics.

Chen Liang, Mohammad Norouzi, Jonathan Berant, Quoc Le, and Ni Lao. 2018. Memory augmented policy optimization for program synthesis and semantic parsing. In *Advances in Neural Information Processing Systems 31 (NeurIPS 2018)*.

Xi Victoria Lin, Richard Socher, and Caiming Xiong. 2020. Bridging textual and tabular data for cross-domain text-to-SQL semantic parsing. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4870–4888, Online. Association for Computational Linguistics.

Vanessa Lopez, Christina Unger, Philipp Cimiano, and Enrico Motta. 2013. Evaluating question answering over linked data. *Journal of Web Semantics*, 21:3–13.

Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, Baltimore, Maryland. Association for Computational Linguistics.

Franck Michel, Catherine Faron-Zucker, and Johan Montagnat. 2019. Bridging the semantic web and NoSQL worlds: Generic SPARQL query translation and application to MongoDB. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems XL*, pages 125–165.

Sewon Min, Danqi Chen, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2019. A discrete hard EM approach for weakly supervised question answering. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2851–2864, Hong Kong, China. Association for Computational Linguistics.

Ankur Parikh, Xuezhi Wang, Sebastian Gehrmann, Manaal Faruqui, Bhuwan Dhingra, Diyi Yang, and Dipanjan Das. 2020. ToTTo: A controlled table-to-text generation dataset. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1173–1186, Online. Association for Computational Linguistics.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32 (NeurIPS 2019)*.

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. Association for Computing Machinery.

Eric Prud'hommeaux and Andy Seaborne. 2008. SPARQL query language for RDF. W3C Recommendation, World Wide Web Consortium.

Ohad Rubin and Jonathan Berant. 2021. SmBoP: Semi-autoregressive bottom-up semantic parsing. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 311–324, Online. Association for Computational Linguistics.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468, New Orleans, Louisiana. Association for Computational Linguistics.

Peng Shi, Patrick Ng, Zhiguo Wang, Henghui Zhu, Alexander Hanbo Li, Jun Wang, Cicero Nogueira dos Santos, and Bing Xiang. 2021. Learning contextual representations for semantic parsing with generation-augmented pre-training. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Alon Talmor and Jonathan Berant. 2018. The web as a knowledge-base for answering complex questions. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 641–651, New Orleans, Louisiana. Association for Computational Linguistics.

Priyansh Trivedi, Gaurav Maheshwari, Mohnish Dubey, and Jens Lehmann. 2017. LC-QuAD: A corpus for complex question answering over knowledge graphs. In *The Semantic Web – ISWC 2017*, pages 210–218.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30 (NIPS 2017)*.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *Advances in Neural Information Processing Systems 28 (NIPS 2015)*.

Bailin Wang, Mirella Lapata, and Ivan Titov. 2021. Learning from executions for semantic parsing. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2747–2759, Online. Association for Computational Linguistics.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7567–7578, Online. Association for Computational Linguistics.

Bailin Wang, Ivan Titov, and Mirella Lapata. 2019. Learning semantic parsers from denotations with latent structured alignments and abstract programs. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3774–3785, Hong Kong, China. Association for Computational Linguistics.

Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a semantic parser overnight. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1332–1342, Beijing, China. Association for Computational Linguistics.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Tomer Wolfson, Mor Geva, Ankit Gupta, Matt Gardner, Yoav Goldberg, Daniel Deutch, and Jonathan Berant. 2020. Break it down: A question understanding benchmark. *Transactions of the Association for Computational Linguistics*, 8:183–198.

Wen-tau Yih, Matthew Richardson, Chris Meek, Ming-Wei Chang, and Jina Suh. 2016. The value of semantic parse labeling for knowledge base question answering. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 201–206, Berlin, Germany. Association for Computational Linguistics.

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.

Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. TaBERT: Pretraining for joint understanding of textual and tabular data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8413–8426, Online. Association for Computational Linguistics.

Tao Yu, Chien-Sheng Wu, Xi Lin Victoria, Wang Bailin, Yi Chern Tan, Xinyi Yang, Dragomir Radev, Richard Socher, and Caiming Xiong. 2021. GraPPa: Grammar-augmented pre-training for table semantic parsing. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

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. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Rui Zhang, Tao Yu, Heyang Er, Sungrok Shim, Eric Xue, Xi Victoria Lin, Tianze Shi, Caiming Xiong, Richard Socher, and Dragomir Radev. 2019. Editing-based SQL query generation for cross-domain context-dependent questions. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5338–5349, Hong Kong, China. Association for Computational Linguistics.

Victor Zhong, Mike Lewis, Sida I. Wang, and Luke Zettlemoyer. 2020. Grounded adaptation for zero-shot executable semantic parsing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6869–6882, Online. Association for Computational Linguistics.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *arXiv*, 1709.00103v7.

Supplementary Material (Appendix) SPARQLing Database Queries from Intermediate Question Decompositions

A QDMR-to-SPARQL translator

Table 8 contains the full list of QDMR operators used in our paper.

Algorithm 1 sketches the QDMR-to-SPARQL translator. It is a recursive procedure that creates SPARQL queries for all QDMR LF steps. At its core, it constructs one or several patterns for the step arguments and then connects them into another pattern in a way specific to the LF operator of the current step.

Importantly, the patterns for LF operators can be of two types: inner (inline) and full. An inner pattern represents the internal part of a query that needs to be placed inside the curly brackets `{...}`. A full pattern corresponds to a full query that can be executed directly (starts with the `SELECT` keyword). An inner pattern can be converted to full by using the `SELECT <output vars> WHERE {<inner>}` construction. The full pattern can be converted to inner by creating a subquery via `{<full>}` (here, the output variables of `<full>` pattern become available in the scope where the subquery is created).

Different LF operators require and produce different patterns: inner of full. Next, we specify a pattern for each LF operator.

The `SELECT` operator adds the grounded object to the context: a self link for a table, a link for a column, a link with a filtering condition for a value.

The `PROJECT` operator creates a context for the argument and does the same as `SELECT`. To connect instances from different columns, we use the breadth-first search to find the shortest path in the undirected graph where all the columns of all tables represent nodes and edges appear between the primary key of each table and all other columns of the same table, and along with the foreign links.

The `COMPARATIVE` operator first creates an inner `<pattern>` for its arguments and then adds a filtering condition from the l.h.s. values `<filter_var>`, the operation `<comparator>` and the r.h.s. value `<value>`:

```
<pattern>
FILTER(<filter_var><comparator><value>).
```

Algorithm 1 QDMR-to-SPARQL translator

```
function QDMR2SPARQL( )
    GetContext([$i_{\text{out}}$], True, $\emptyset$)

function GetContext(indices, inline, $C$)
    if C is empty then $C \leftarrow$ InitC( )
    while not too many tries do
        for $i$ in indices do
            $C \leftarrow$ AddIndex($i$, inline, $C$)
    return C

function AddIndex($i$, inline, $C$)
    op $\leftarrow$ QDMR op at step #$i$
    args $\leftarrow$ get indices of arguments of step #$i$
    needs_inl $\leftarrow$ [op needs inline args]
    makes_inl $\leftarrow$ [op makes inline context]
    $C \leftarrow$ GetContext(args, needs_inl, $C$)
    $C \leftarrow$ fill the pattern of op
    if inline $\neq$ makes_inl then
        $C \leftarrow$ convert $C$ to inline/full
    return $C$
```

The `AGGREGATE` operator computes the aggregator `<agg_op>` from a set of values. This operator takes the inner pattern `<pattern>` as input (with `<var>` correspondings to the set of values to aggregate) and produces the full query with the output variable `<output_var>` as the output:

```
SELECT (<agg_op>(<var>) as <output_var>)
WHERE { <pattern> }
```

The `SUPERLATIVE` operator filters the instances such that some related attribute has the min/max value. The operator first computes the min/max value with a built-in `AGGREGATE` operator then filters (similar to `COMPARATIVE`) the patterns based on the computed value:

```
{SELECT (<agg_op>(<var>) AS <minmax_var>)
  WHERE { <pattern_inner> } }
<pattern_outer>
FILTER(<query_outer_var>=<minmax_var>).
```

The `SUPERLATIVE` operator requires two inner patterns as input `<pattern_inner>`, `<pattern_outer>` and makes an inner pattern as the output.

The `GROUP` operator groups the values `<var>` by the equal values of the related attribute `<index_var>`:

```
SELECT (<agg_op>(<var>) AS <output_var>)
WHERE { <pattern> }
GROUP BY <index_var>
```

Operator	Arguments	Type	Description
SELECT	subject distinct	text bool	Select subject (possibly distinct values)
PROJECT	projection subject distinct	text ref bool	Select projection related to subject (possibly distinct values)
COMPARATIVE	subject attr comparator value distinct	ref ref choice text/ref bool	Select subject such that related attr compares (using $=, \neq, >, <, \geq, \leq$, like) to value (possibly distinct values)
SUPERLATIVE	subject attr operator	ref ref choice	Select subject such that related attr has max/min values
AGGREGATE	subject operator	ref choice	Compute max/min/sum/count of subject
GROUP	subject attr operator	ref ref choice	Group instances of subject such that attr has same values (aggregate with max/min/sum/count)
UNION	ref1, ref2, etc.	ref	Get the union of ref1, ref2, etc.
INTERSECT	subject attr1, attr2	ref ref	Get instances of subject related to both attr1 and attr2
DISCARD	subject minus	ref ref	Get instances of subject excluding instances of minus
SORT	subject attr direction	ref ref choice	Order instances of subject such that related attr is ordered in asc/desc direction

Table 8: QDMR operators, their arguments, types of the arguments. The full version of Table 2.

The aggregation is done with the operator <agg_op>. The input pattern <pattern> is inner, and the output is the full pattern with the output variable <output_var>.

The UNION operator can actually correspond to several operators: horizontal union, vertical union, union of aggregators, union after group. By horizontal union, we mean the union of two or more related variables from the same pattern. These variables have to correspond to different database columns. By vertical union, we mean the union of two or more variables corresponding to the same column but coming from different patterns. This case is implemented with the UNION keyword from SPARQL using the following construction:

```
{ <pattern1> }
UNION
{ <pattern2> }
```

The union-after-group case is a special but common situation when arguments contain the result of the GROUP operator and the index variable of the same operator. We implement this case similar to the pattern of the GROUP operator but with several variables in the output. The union of aggregators is another common special case when the arguments of the UNION contain several aggregators from the same pattern. We simply output these several aggregators by concatenating them after the SPARQL SELECT keyword.

The INTERSECT operator effectively consists in sequentially applying two COMPARATIVE operators that do not have explicit comparisons as arguments.

The DISCARD operator is based of the pattern very similar to the vertical union:

```
{ <pattern1> }
MINUS
{ <pattern2> }
```

The SORT operator consists in adding the ORDER BY keyword at the end of the full pattern:

```
SELECT <output_vars>
WHERE { <pattern> }
ORDER BY ASC/DESC(<index_var>)
```

B Implementation details

We implemented our model on the top of the RAT-SQL code[7] built with Pytorch (Paszke et al., 2019). We use pretrained BERT and GraPPa from the Transformers library (Wolf et al., 2020). To support SPARQL queries and RDF databases, we used two libraries: RDFLib[8] and the open-source version of the Virtuoso system.[9] RDFLib was much easier to install (a python package), but Virtuoso allowed to run SPARQL queries on pre-loaded databases much faster.

To choose relevant values from a database, we tokenized question and all unique database values using the Stanford CoreNLP library (Manning et al., 2014), filtered tokens using NLTK[10] English stopwords, and then picked top-25 values with higher similarity scores calculated as follows:

- for a numeric value, we gave the maximum score if it exactly matched with some question token, otherwise, we gave the minimum score;
- for other tokens, we gave the maximum score if the value and question stems were the same (we used the Porter and Snowball stemmers from NLTK), otherwise, we calculated similarity score based on the longest continuous matching subsequence (we used Python SequenceMatcher class).

For the neural network architecture and training, we used the same hyperparameters as RAT-SQL (Wang et al., 2020): 8 RAT layers, each with 8 heads and the hidden dimension of 256, 1024 and 512 in self-attention, position-wise feed-forward network and decoder LSTM, respectively. We trained the model with the Adam optimizer (Kingma and Ba, 2014) and polynomial decay scheduler used by Wang et al. (2020). The batch size was 24, the overall number of iterations was 81000 for all models.

The training time on 4 NVIDIA V100 GPUs was approximately 24 hours.

C QDMR grammar

```
root           ⟶ step
step           ⟶ select, project, sort,
                  group, aggregate,
                  comparative, superlative,
                  intersection, discard,
                  union, final
select         ⟶ distinct, grounding, step
project        ⟶ distinct, project_1arg
                  ref, step
comparative    ⟶ distinct, ref, ref,
                  comp_3arg, step
superlative    ⟶ superlative_op
                  ref, ref, step
group          ⟶ agg_type, ref, ref, step
aggregate      ⟶ agg_type, ref, step
intersection   ⟶ ref, ref, step
discard        ⟶ ref, ref, step
union          ⟶ ref, ref, step
sort           ⟶ ref, ref, order, step
project_1arg   ⟶ grounding | none
comp_3arg      ⟶ comp_op_type,
                  column_type, comp_val
comp_op_type   ⟶ comparative_op | no_op
column_type    ⟶ grounding | no_column
comp_val       ⟶ grounding | ref
comparative_op ⟶ ≠ | > | < | ≥ | ≤ |like
superlative_op ⟶ min | max
order          ⟶ asc | desc
agg_type       ⟶ aggregate_op | grounding
aggregate_op   ⟶ Count | Sum | Avg |
                  Min | Max
```

D Restrictions in the decoding process

The decoding process at the inference stage is sequential, and at each step, there is a set of eligible choices. These sets are always non-empty and are formed using the following restrictions:

- The eligible choices of grounding as aggregate type (`agg_type` ⟶ `grounding`) columns;
- The eligible choices of grounding as the column type in comparative (`column_type` ⟶ `grounding`) are columns with the types from the set of input value types;
- After the model chooses a column in comparative, the eligible choices of grounding as comparative value (`comp_val` ⟶ `grounding`) are the values from this column or with the same type but not from the database;
- After the model chose to skip column (`no_column`) in comparative, the eligible choices of grounding as comparative value (`comp_val` ⟶ `grounding`) are the values not from the database.

[7] https://github.com/microsoft/rat-sql

[8] https://github.com/RDFLib/rdflib

[9] https://github.com/openlink/virtuoso-opensource

[10] https://www.nltk.org/index.html