DBPal: A Fully Pluggable NL2SQL Training Pipeline

Nathaniel Weir[1] Prasetya Utama[2] Alex Galakatos[3] Andrew Crotty[3]
Amir Ilkhechi[3] Shekar Ramaswamy[3] Rohin Bhushan[3] Nadja Geisler[2]
Benjamin Hättasch[2] Steffen Eger[2] Ugur Cetintemel[3] Carsten Binnig[2]

[1] Johns Hopkins University {nweir3@jhu.edu} [2]TU Darmstadt {first.last@cs.tu-darmstadt.de}
[3]Brown University {first_last@brown.edu}

ABSTRACT

Natural language is a promising alternative interface to DBMSs because it enables non-technical users to formulate complex questions in a more concise manner than SQL. Recently, deep learning has gained traction for translating natural language to SQL, since similar ideas have been successful in the related domain of machine translation. However, the core problem with existing deep learning approaches is that they require an enormous amount of training data in order to provide accurate translations. This training data is extremely expensive to curate, since it generally requires humans to manually annotate natural language examples with the corresponding SQL queries (or vice versa).

Based on these observations, we propose DBPal, a new approach that augments existing deep learning techniques in order to improve the performance of models for natural language to SQL translation. More specifically, we present a novel training pipeline that automatically generates synthetic training data in order to (1) improve overall translation accuracy, (2) increase robustness to linguistic variation, and (3) specialize the model for the target database. As we show, our DBPal training pipeline is able to improve both the accuracy and linguistic robustness of state-of-the-art natural language to SQL translation models.

ACM Reference Format:
Nathaniel Weir et al. 2020. DBPal: A Fully Pluggable NL2SQL Training Pipeline. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3380589

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3380589

1 INTRODUCTION

In order to effectively leverage their data, DBMS users are required to not only have prior knowledge about the database schema (e.g., table and column names, entity relationships) but also a working understanding of the syntax and semantics of SQL. Unfortunately, despite its expressiveness, SQL can often hinder non-technical users from exploring and making use of data stored in a DBMS. These requirements set "a high barrier to entry" for data exploration and have therefore triggered new efforts to develop alternative interfaces that allow non-technical users to explore and interact with their data conveniently.

For example, imagine that a doctor wants to look at the age distribution of patients with the longest stays in a hospital. To answer this question, the doctor would either need to write a complex nested SQL query or work with an analyst to craft the query. Even with a visual exploration tool (e.g., Tableau [1], Vizdom [12]), posing such a query is nontrivial, since it requires the user to perform multiple interactions with an understanding of the nested query semantics. Alternatively, with a natural language (NL) interface, the query is as simple as stating: "What is the age distribution of patients who stayed longest in the hospital?"

Based on this observation, a number of Natural Language Interfaces to Databases (NLIDBs) have been proposed that aim to translate natural language to SQL (NL2SQL). The first category of solutions are rule-based systems (e.g., NaLIR [25, 26]), which use fixed rules for performing translations. Although effective in specific instances, these approaches are brittle and do not generalize well without substantial additional effort to support new use cases. More recently, deep learning techniques [22, 43, 44] have gained traction for NL2SQL, since similar ideas have achieved success in the related domain of machine translation. For example, generic sequence-to-sequence (seq2seq) [51] models have been successfully used in practice for NL2SQL translation, and more advanced approaches like SyntaxSQLNet [46], which augments deep learning models with a structured model that considers the syntax and semantics of SQL, have also been proposed.

However, a crucial problem with deep learning approaches is that they require an enormous amount of training data in order to build accurate models [21, 38]. The aforementioned approaches have largely ignored this problem and assumed the availability of large, manually-curated training datasets (e.g., using crowdsourcing). In almost all cases, however, gathering and cleaning such data is a substantial undertaking that requires a significant amount of time, effort, and money.

Moreover, existing approaches for NL2SQL translation attempt to build models that generalize to new and unseen databases, yielding performance that is generally decent but does not perform as well as running new queries on the databases used for training. That is, the training data used to translate queries for one specific database, such as queries containing words and phrases pertaining to patients in a hospital, does not always allow the model to generalize to queries in other domains, such as databases of geographical locations or flights.

In order to address these fundamental limitations, we propose DBPal, a fully pluggable NL2SQL training pipeline that can be used with any existing NL2SQL deep learning model to improve translation accuracy. DBPal implements a novel training pipeline for NLIDBs that synthesizes its training data using the principle of *weak supervision* [11, 15].

The basic idea of weak supervision is to leverage various heuristics and existing datasets to automatically generate large (and potentially noisy) training data instead of manually handcrafting training examples. In its basic form, only the database schema is required as input to generate a large collection of pairs of NL queries and their corresponding SQL statements that can be used to train any NL2SQL deep learning model.

In order to maximize our coverage across natural linguistic variations, DBPal also uses additional input sources to automatically augment the training data through a variety of techniques. One such augmentation step, for example, is an automatic paraphrasing process using an off-the-shelf paraphrasing database [29]. The goal of these augmentation steps is to make the model robust to different linguistic variations of the same question (e.g., "What is the age distribution of patients who stayed longest in the hospital?" and "For patients with the longest hospital stay, what is the distribution of age?").

In our evaluation, we show that DBPal, which requires no manually crafted training data, can effectively improve the performance of a state-of-the-art deep learning model for NL2SQL translation. Our results demonstrate that an NLIDB can be effectively bootstrapped without requiring manual training data for each new database schema or target domain. Furthermore, if manually curated training data is available, such data can still be used to complement our proposed data generation pipeline.

In summary, we make the following contributions:

- We present DBPal, a fully pluggable natural language to SQL (NL2SQL) training pipeline that automatically synthesizes training data in order to improve the translation accuracy of an existing deep learning model.
- We propose several data augmentation techniques that give the model better coverage and make it more robust towards linguistic variation in NL queries.
- We propose a new benchmark that systematically tests the robustness of a NLIDB to different linguistic variations.
- Using a state-of-the-art deep learning model, we show that our training pipeline can improve translation accuracy by up to almost 40%.

The remainder of this paper is organized as follows. First, in Section 2, we introduce the overall system architecture of DBPal. Next, in Section 3, we describe the details of DBPal's novel training pipeline, which is based on weak supervision. We then show how the learned model for NL2SQL translation is applied at runtime in Section 4. Furthermore, we discuss the handling of more complex queries like joins and nested queries in Section 5. In order to demonstrate the effectiveness of DBPal, we present the results of our extensive evaluation in Section 6. Finally, we discuss related work in Section 7 and then conclude in Section 8.

2 OVERVIEW

In the following, we first discuss the overall architecture of a NLIDB and then discuss DBPal, our proposed training pipeline based on weak supervision that synthesizes the training data from a given database schema.

2.1 System Architecture

Figure 1 shows an overview of the architecture of our fully functional prototype NLIDB, which consists of multiple components, including a user-interface that allows users to pose NL questions that are automatically translated into SQL. The results from the user's NL query are then returned to the user in an easy-to-read tabular visualization.

At the core of our prototype is a *Neural Translator*, which is trained by DBPal's pipeline, that translates incoming NL queries coming from a user into SQL queries. Importantly, our fully pluggable training pipeline is agnostic to the actual translation model; that is, DBPal is designed to improve the accuracy of existing NL2SQL deep learning models (e.g., SyntaxSQLNet [46]) by generating training data for a given database schema.

2.1.1 Training Phase. During the training phase, DBPal's training pipeline provides existing NL2SQL deep learning models with large corpora of synthesized training data. This

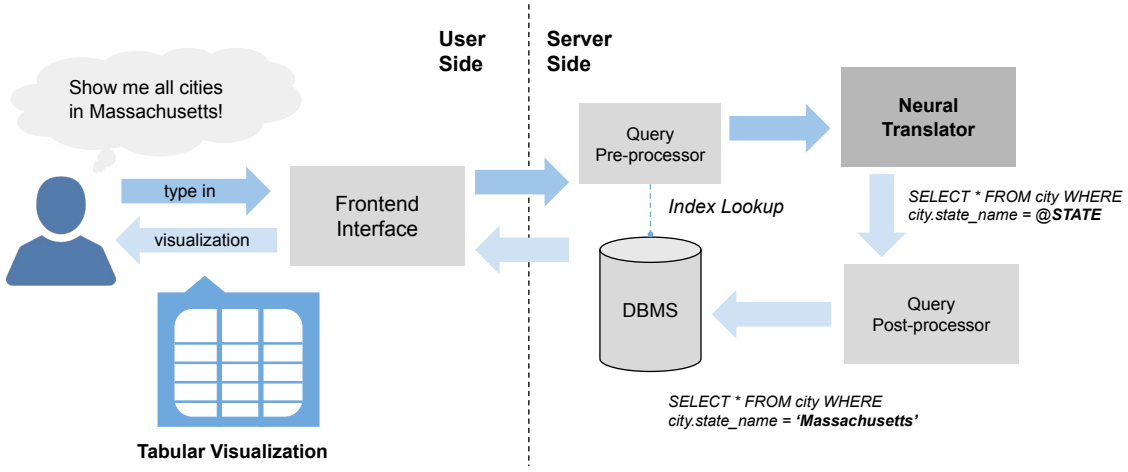

Figure 1: Lifecycle of a NL query through a Neural Translator trained by DBPal's training pipeline

training pipeline, described further in Section 2.2.1, consists of three steps to synthesize the training data: (1) generator, (2) augmentation, and (3) lemmatizer. Once training data is synthesized by DBPal's pipeline, it can then be used (potentially together with existing manually curated training data) to train existing neural translation models that can be plugged into the training pipeline.

2.1.2 Runtime Phase. The runtime phase can leverage a model (Neural Translator) that was trained by DBPal, as shown on the right-hand side of Figure 2. The *Parameter Handler* is responsible for replacing the constants in the input NL query with placeholders to make the translation model independent from the actual database and help to avoid retraining the model if the underlying database is updated. For example, for the input query shown in Figure 2 (i.e., *"What are cities whose state is Massachusetts?"*), the *Parameter Handler* replaces *"Massachusetts"* with the appropriate schema element using the placeholder `@STATE`. The *Lemmatizer* then combines different variants of the same word to a single root. For example, the words *"is"*, *"are"*, and *"am"* are all mapped to the root word *"be"*. Then, the *Neural Translator* works on these anonymized NL input queries and creates output SQL queries, which also contain placeholders. In the example shown in Figure 2, the output of the Neural Translator is: `SELECT name FROM cities WHERE state = @STATE`. Finally, the *Post-processor* replaces the placeholders with the actual constants such that the SQL query can be executed.

2.2 Training Pipeline

The basic flow of the training pipeline is shown on the left-hand side of Figure 2. In the following, we describe the training pipeline and focus in particular on the data generation framework. The details of the full training pipeline are explained further in Section 3.

2.2.1 Generator. In the first step, the *Generator* uses the database schema along with a set of seed templates that describe typical NL-SQL pairs to generate an initial training set. In the second step, *Augmentation*, the training data generation pipeline then automatically adds to the initial training set of NL-SQL pairs by leveraging existing general-purpose data sources and models to linguistically modify the NL part of each pair.

The main idea is that each seed template covers a typical class of SQL queries (e.g., a `SELECT-FROM-WHERE` query with a simple predicate). Composing the seed templates is only a minimal, one-time overhead, and all templates are independent of the target database (i.e., they can be reused for other schemas). Furthermore, in DBPal, we assume that the database schema provides human-understandable table and attribute names, but the user can optionally annotate the schema to provide more readable names if required; deriving readable schema names automatically is an orthogonal issue.

The schema information is then used to instantiate these templates using table and attribute names. Additionally, manually predefined dictionaries (e.g., to cover synonyms) can be used to instantiate simple variations of NL words and phrases (e.g., "Show me" and "What is" for the `SELECT` clause). Currently, DBPal contains approximately 100 seed templates. A typical training set that can be generated from these templates contains around 1 million NL-SQL pairs for a simple, single-table database schema and around 2-3 million for more complicated schemas.

2.2.2 Augmentation. A core aspect of our pipeline is the *Augmentation* step that automatically expands the training

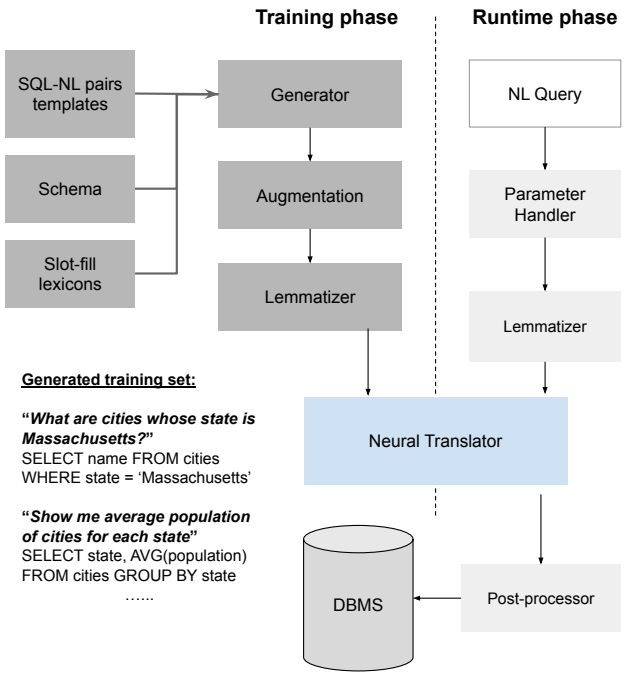

Figure 2: DBPal's Training and Runtime Phases

data produced by our Generator in order to offer more accurate and linguistically robust translations. During augmentation, the training data generation pipeline automatically adds new NL-SQL pairs by leveraging existing general-purpose data sources and models to linguistically vary the NL part of each pair. The goal of the augmentation phase is thus to cover a wide spectrum of linguistic variations for the same SQL query, which represent different versions of how users might phrase the query in NL. This augmentation is the key to make the translation model robust and allows DBPal to provide better query understanding capabilities than existing standalone approaches. Section 3.2 describes this process in more detail.

2.2.3 Lemmatization. Finally, in the last step of the data generation procedure, the resulting NL-SQL pairs are lemmatized to normalize the representation of individual words. During this process, different forms of the same word are mapped to the word's root in order to simplify the analysis (e.g., *"cars"* and *"car's"* are replaced with *"car"*). The same lemmatization is applied at runtime during the aforementioned pre-processing step.

3 TRAINING PHASE

In this section, we describe DBPal, our fully pluggable training data generation pipeline, which is designed to improve the translation accuracy and linguistic robustness of existing NL2SQL deep learning models. After describing the steps of our training pipeline in detail, we discuss an optimization procedure of the data generation process to increase the model quality via parameter tuning. Finally, we elaborate on the model training process, including a description of the details of the model architecture and the hyperparameters used in training.

3.1 Data Instantiation

The main observation of the instantiation step is that SQL, as opposed to NL, has significantly less expressivity. We therefore use query templates to instantiate different possible SQL queries that a user might phrase against a given database schema, such as:

Select {Attribute}(s) From {Table} Where {Filter}

The main idea of data instantiation is that the space of possible SQL queries a user might phrase against a given database schema can be defined using a set of SQL templates. The SQL templates cover a variety of query types, from simple `SELECT-FROM-WHERE` queries to more complex group-by aggregation queries, as well as some simple nested queries. For each SQL template, we define one or more NL templates as counterparts for direct translation, such as:

{SelectPhrase} {Attribute}(s) {FromPhrase} {Table}(s) {WherePhrase} {Filter}

It is important to note that we do not use actual constants in the filter predicates. Instead, we use placeholders (e.g., *@AGE*) that represent an arbitrary constant for a given table attribute. This makes the model trained on the generated data independent of concrete values used in the database; thus retraining is not required after inserts or updates.

To account for the expressivity of NL compared to SQL, our templates contain slots for speech variation (e.g., *SelectPhrase*, *FromPhrase*, *WherePhrase*) in addition to slots for database objects (e.g., tables, attributes). Then, to instantiate the initial training set, the Generator repeatedly instantiates each of our NL templates by filling in the corresponding slots. Table, column, and filter slots are filled using information from the database's schema, while a diverse array of NL slots are filled using manually crafted dictionaries of synonymous words and phrases. For example, the phrases *"what is"* or *"show me"* can be used to instantiate the *SelectPhrase*. A fully instantiated NL-SQL pair might look like:

Table, column, and filter slots are filled using information from the database's schema, while a diverse array of natural language slots are filled using manually crafted dictionaries of synonymous words and phrases. For example, the phrases

"what is" or *"show me"* can be used to instantiate the *SelectPhrase*. A fully instantiated NL-SQL pair might look like:

Instantiated NL:
Show the names of all patients with age 20.

Instantiated SQL:
`SELECT name FROM patients WHERE age = 20`

An important part of training data instantiation is balancing the number of NL-SQL pairs that are instantiated per template. If we naively replace the slots of a query template with all possible combinations of slot instances (e.g., all attribute combinations from the schema), then instances that result from templates with more slots would dominate the training set and bias the model. More specifically, an imbalance of instances can lead to a biased training set where the model would prefer certain translations over others only due to the fact that certain variations appear more often. We therefore randomly sample from the possible instances to get a good coverage of different queries and to keep the number of instances per query template balanced.

Finally, for each initial NL template, we additionally provide some manually curated paraphrased NL templates that follow particular paraphrasing techniques [41], covering categories such as syntactical, lexical, and morphological paraphrasing. The existence of multiple corresponding NL templates for each SQL template allows us to systematically cover a range of possible linguistic variations. Importantly, the paraphrased templates can be applied to instantiate the training data for any given schema, and the instantiated NL-SQL pairs are also automatically paraphrased during automatic data augmentation.

For example, consider the following paraphrased NL template and a corresponding instantiation:

Paraphrased NL Template:
For {Table}(s) with {Filter}, what is their {Attribute}(s)?

Instantiated NL Template:
For patients with age @AGE, what is their name?

We further expand upon our augmentation techniques in the following section.

3.2 Data Augmentation

Unsurprisingly, numerous ways exist to express the same idea in NL. For example, the questions *"Show me the names of all patients older than 18"* and *"What are the names of patients who have an age greater than 18?"* are semantically equivalent. Therefore, to make the NL2SQL model more robust to these linguistic variations, we apply the following augmentation steps for each instantiated NL-SQL pair.

3.2.1 Automatic Paraphrasing. First, we augment the training set by generating duplicate NL-SQL pairs. This process involves randomly selecting words/subphrases of the NL query and paraphrasing them using the Paraphrase Database (PPDB) [29] as a lexical resource, for example:

Input NL Query:
Show the names of all patients with age @AGE.

PPDB Output:
demonstrate, showcase, display, indicate, lay

Paraphrased NL Query:
Display the names of all patients with age @AGE.

PPDB is an automatically extracted database containing millions of paraphrases in 27 different languages. DBPal uses PPDB's English corpus, which provides over 220 million paraphrase pairs consisting of 73 million phrasal and 8 million lexical paraphrases, as well as 140 million paraphrase patterns, which capture a wide range of meaning-preserving syntactic transformations. The paraphrases are extracted from bilingual parallel corpora totaling over 100 million sentence pairs and over 2 billion English words.

During paraphrasing, we randomly replace words and subphrases of the input NL query with available paraphrases provided by PPDB. For example, searching in PPDB for a paraphrase of the word *enumerate*, as in *"Enumerate the names of patients with age 80"*, we get suggestions such as *"list"* or *"identify"* as alternatives.

An important question is how aggressively to apply automatic paraphrasing. We therefore provide two parameters to tune the automatic paraphrasing in DBPal. The first parameter $size_{para}$ defines the maximum size of the subclauses (in number of words) that should be replaced in a given NL query. A second parameter num_{para} defines the maximum number of paraphrases that are generated as linguistic variations for each subclause. For example, setting $size_{para} = 2$ will replace subclauses of size 1 and 2 (i.e., unigrams and bigrams) in the input NL query with paraphrases found in PPDB. Furthermore, setting $num_{para} = 3$, each of the unigrams and bigrams will be replaced by at most 3 paraphrases.

Setting these two parameters in an optimal manner is, however, not trivial: if we set both parameters to high values, we can heavily expand our initial training set of NL-SQL query pairs using many different linguistic variations, which hopefully will increase the overall robustness of DBPal. At the same time, we might also introduce noise into the training dataset, since PPDB also includes some paraphrases that are of low quality.

DBPal has default values for all of these parameters that we have empirically determined to have the best performance

on multiple database schemas, which are used in our evaluation in Section 6. In order to optimize these parameters to increase the overall model accuracy for a given input database schema, we provide an optimization procedure that we discuss in Section 3.3. Our procedure automatically finds a parameterization of the data generator that balances, among others, the trade-off between these two dimensions: size of the augmented training data and noise in the training data.

3.2.2 Missing Information. Another challenge of input NL queries is missing or implicit information. For example, a user might ask for *"patients with influenza"* instead of *"patients diagnosed with influenza"*, where the referenced attribute (i.e., *"diagnosis"*) is never explicitly stated.

Therefore, to make the translation more robust to missing or implicit context, we randomly drop words and subphrases from the NL training queries. For example, from the sentence *"patients diagnosed with influenza"*, DBPal might decide to drop the word *"diagnosed"*, allowing the translation model to be able to successfully answer the question *"Who are the patients with influenza?"*

Similar to paraphrasing, an interesting question is: which words or subphrases should be removed and how frequently should we remove them? Again, aggressively removing words increases the training data size, since more variations are generated. On the other hand, however, we might introduce noisy training data that leads to a drop in translation accuracy and, counterproductively, produces less linguistically robust models.

In order to tune how aggressively we drop words and subphrases, we follow a similar protocol as the paraphrasing process by randomly selecting words in the NL query and removing them in a duplicate. Thus, we additionally introduce a parameter named $num_{missing}$ that defines the maximum number of query duplicates with removed words for a given input NL query. We also include a parameter $randDrop_p$ that defines how often the generator will choose to remove words from a particular NL query at all. Analogously to automatic paraphrasing, we set these two parameters for a given input database schema automatically using the procedure described in Section 3.3.

3.2.3 Other Augmentations. For the automatic data augmentation, we apply some additional techniques to increase the linguistic variations. One example is the use of available linguistic dictionaries for comparatives and superlatives. For example, by using these resources, we can replace the general phrase *greater than* in an input NL query by *older than* if the domain of the schema attribute is set to *age*.

In the future, we plan on extending our augmentation techniques in a variety of ways. For example, one possible avenue is to enhance our automatic paraphrasing using other language sources and not only PPDB. We also plan to investigate the idea of using an off-the-shelf part-of-speech tagger to annotate each word in a given NL query. These annotations can be used in different forms (e.g., we could use them in the automatic paraphrasing to identify better paraphrases or to infer a request for a nested query). Another idea is to use part-of-speech tags to apply the word removal only for certain classes of words.

3.3 Optimization Procedure

One important challenge of the automatic data generation steps is to instantiate the training data such that the translation model will provide a high accuracy. For example, the template-based training data instantiation step also has parameters that can be tuned to control the number of basic NL-SQL pairs that are instantiated for each template. Without tuning these parameters, the data generation process could introduce bias in the generated training data based on a given schema if we exhaustively generate all possible NL-SQL pairs. Furthermore, the augmentation steps require several parameters for each step that define how aggressively paraphrasing and removing information is applied to an input NL query.

We therefore attempt to automatically optimize the configuration of the generator parameters given a particular database schema. The intuition behind this strategy comes from observations made about the translation model's behavior. In particular, we note that models are typically very susceptible to overfitting to over-represented NL-SQL queries. For example, if we overpopulate the training set with the SQL *count* queries (the natural language parallel will usually include words like *"how many"*), the model will likely output a *count* query for all aggregations simply because it sees particular NL words that most commonly appeared with the word *count* during training. Queries like *"How large is the area of Alaska?"* might be therefore be mapped to a *count* instead of *sum* simply for this reason.

Table 1 lists all parameters that are available in DBPal to tune the data generation process and explains their meanings. As mentioned before, these parameters define the main characteristics of the training data instantiation and augmentation steps, and thus they have an effect on the accuracy and robustness of the translation model. In order to find the optimal parameter values of the data generation process for a given schema automatically, we model the data generation procedure as the following function:

$$Acc = Generate(D, T, \phi)$$

The inputs of this function are the database D that describes the schema and contains some sample data, a test workload T of input NL queries and expected output SQL

Parameter	Explanation
Data Instantiation	
$size_{slotfills}$	Maximum number of instances created for a NL-SQL template pair using slot-filling dictionaries.
$size_{tables}$	Maximum number of tables supported in join queries.
$groupBy_p$	Probabilities of generating a *GROUP BY* version of a generated query pair.
$join_{boost}$, agg_{boost}, $nest_{boost}$	Control the balance of various types of SQL statements relative to each other and the number of templates used.
Data Augmentation	
$size_{para}$	Maximum size of subclauses that are automatically replaced by a paraphrase.
num_{para}	Maximum number of paraphrases that are used to vary a subclause.
$num_{missing}$	Maximum number of words that are removed for a given input NL query.
$randDrop_p$	Probability of randomly dropping words from a generated query.

Table 1: Parameters of the Data Generation Procedure

queries, as well as a possible instantiation of all the tuning parameters ϕ listed in the Table 1. The output of the generation procedure *Acc* is the accuracy of our model that is trained on the generated dataset using D as well as ϕ and then evaluated using the test workload T. It is important to note that we can either use a test workload T that is created automatically by using a random sample of the generated training data (i.e., we split the test set from the training set) or by providing a small representative set of NL-SQL query pairs that are curated manually.

The goal of the optimization procedure is to find a parameter set ϕ that maximizes the accuracy *Acc*. Automatic optimization techniques are useful for global optimization problems that evaluate expensive black-box functions; as such it has become popular for optimizing deep learning models that take in a seemingly arbitrary set of hyperparameters, such as the number of layers or perceptrons per layer of a convolutional neural network (CNN). However, instead of applying the optimization procedure to our translation model, we extrapolate one step backwards and attempt to optimize the nature of the training set to which the model will be exposed.

In machine learning, there exist several strategies for automatically tuning hyperparameters. In DBPal, we use a random search approach to automatically tune the hyperparameters ϕ of the function *Generate*. For each candidate set of parameters, the entire system pipeline, including data generation and model training (labeled $Generate(D, T, \phi)$), is completed and the accuracy is returned. Random search is a standard technique for hyperparameter-tuning and differs from grid search, which is an alternative to random search, mainly in that it searches the specified subset of hyperparameters randomly instead of exhaustively.

The major benefit of random search is the reduced runtime in practice to find a set of hyperparameters that increases the accuracy of the learned model. However, unlike grid search, with random search we are not guaranteed to find the optimal combination of hyperparameters. In the experimental evaluation, we show that by using random search, we can find parameters for the data generation process to produce training data that can provide a high accuracy for the trained model. We also experimented with more sophisticated hyperparameter search strategies like Bayesian optimization, which did not find to improve the accuracy over the random search strategy.

3.4 Neural Translation Model

As previously mentioned, DBPal is fully pluggable and is designed to improve the accuracy of any existing NL2SQL deep learning model. Therefore, importantly, existing models, ranging from simple seq2seq to more complex ones like SyntaxSQLNet [46], can be used for the translation and still benefit from our proposed training pipeline. Additionally, since a great deal of ongoing work is currently focused on producing better NL2SQL models, our approach is similarly able to improve the performance of any new advancements that the NL community develops for translation. Since our main contribution of this work is the novel data generation approach, a detailed discussion of deep model architectures is beyond the scope of this paper.

4 RUNTIME PHASE

In this section, we describe the query translation pipeline. The complete process of the runtime phase is shown in Figure 2 (right-hand side). From the given input NL query to the output SQL query, three major processing phases are performed: pre-processing, query translation, and post-processing. The output SQL query is then executed against the database and the result is returned to the user interface in tabular form, as shown in Figure 1.

4.1 Pre-Processing and Query Translation

The input to the pre-processing step is a NL query formulated by the user, such as the following:

User NL Query (with constants):
Show me the name of all patients with age 80

As previously mentioned, during pre-processing, parameter values (i.e., constants) are replaced with special placeholders. This step is performed to translate queries independently from the database content. The resulting intermediate query is as follows:

Input NL Query (without constants):
Show me the name of all patients with age @AGE
Output SQL Query (without constants):
`SELECT name FROM patient WHERE age=@AGE`

Replacing the constants in the input NL query with their placeholders is a nontrivial task. The process might not be deterministic, since the same constant might map to different columns. This sub-task, commonly referred to as "variable anonymization," has been identified by other groups as an important challenge in the NL2SQL pipeline. In their work towards systematic benchmarking for NL2SQL systems, [19] acknowledge that anonymization can be treated as a separate task, and provide benchmarks with and without having already performed the anonymization. As such, our paper follows the former setup and evaluates on test sets with pre-anonymized values.

In practice, as a temporary solution in the basic version of DBPal, we build an index on each attribute of the schema that maps constants to possible attribute names. Moreover, the user might have provided a string constant in the input NL query that is only similar to the one used in the database (e.g., the user provides "New York City" instead of "NYC"). In the current version of DBPal, we use a similarity function to replace constants with their most similar value that is used in the database. We therefore search the possible column values and compute a string similarity metric with the string constant provided by the user. In our prototype, we currently use the Jaccard index, but the function can be replaced with any other similarity metric. In cases where the similarity of all values for the user-specified string is too low (which could mean that the value does not exist in the database), we use the constant as given by the user and do not replace it.

Finally, as a last step of pre-processing, we lemmatize the input NL query to normalize individual words and thus increase the similarity of the training data (which we also lemmatize) and the input NL query the user provides at runtime. After all pre-processing steps are applied, the trained model is used to map the anonymized and lemmatized NL query into an output SQL query, as shown previously.

4.2 Post-Processing

After pre-processing and translation, a post-processing phase is applied. First, the placeholders in the output SQL query are replaced by the appropriate database constants. Then, we use SQL syntax knowledge to repair potential translation errors of the model.

The first step is simply the inverse step of the pre-processing phase. For example, the placeholder in the output SQL query shown before should be replaced by the according constant that was present in the user input:

Output SQL Query (with constants):
`SELECT name FROM patient WHERE age=80`

Hence, we need to replace the placeholder in the SQL output of the model with the constant used in the input NL query (e.g., @AGE is replaced by 80 in the example above).

In the second step of the post-processing phase, DBPal uses knowledge about the SQL syntax to repair potential translation errors that might result from applying the model. One typical example is that the attributes used in the output SQL query and the table names do not match (e.g., the query asks for patient names but the table patient is not used in the `FROM` clause). In this case, the post-processing step adds missing tables to the `FROM` clause. The most likely join path is selected from the schema using the shortest join path between the table already present in the `FROM` clause and the missing table. This is similar to the general join handling, which we discuss in detail in the next section.

5 COMPLEX QUERIES

In the previous sections, we have shown both the training and runtime phase of DBPal for example queries with single tables. In this section, we discuss how we extend these techniques to handle joins and nested queries as well.

5.1 Join Queries

In order to handle NL input queries that require a join, we extend the template-based instantiation during the training phase such that the attribute slots of a query can be filled with attribute names from multiple tables in the same instance. Attribute slots can be present in different parts of a query (e.g., the `SELECT` or `WHERE` clause). The maximum number of distinct tables that are used during slot-filling can be defined using a parameter called $size_{tables}$, which is a tuning parameter of the data generation process, as previously discussed. Furthermore, we also change the instantiation of table names in the generated SQL query. Instead of enumerating all required tables in the `FROM` clause, we add a special placeholder *@JOIN*. An example for an instantiated NL-SQL pair that use a join might look as follows:

SQL Query (Training Set):
`SELECT AVG(patient.age) FROM @JOIN WHERE doctor.name=@DOCTOR.NAME`
NL Query (Training Set):
What is the average age of patients treated by doctor @DOCTOR.NAME

At runtime, our translation model then outputs a SQL query with a *@JOIN* placeholder when it sees an input NL query with attributes from multiple tables (i.e., it outputs a SQL query without concrete table names in the `FROM` clause). The *@JOIN* placeholder is then replaced in the post-processing step with the actual table names and the join path that contains all tables required by the query. From experience, we observe that this reduces the overall model complexity, since the model does not need to predict actual table names for the `FROM` clause.

Furthermore, as explained before in Section 4, for single-table queries our translation model sometimes produces erroneous SQL queries where the table name in the `FROM` clause does not match the attribute names used. These errors are handled in the post-processing step, where we must infer the correct table names from the attributes used in the SQL query. Thus, increasing the model complexity to predict both the join paths and table names increases the rate of errors that would need to be handled in the post-processing phase. The introduction of the `JOIN` placeholder rectifies these issues.

DBPal's post-processing phase uses the schema information to infer table names and a join path from the attributes in the SQL output of the model. In case multiple join paths are possible to connect all the required tables, we select the join path that is minimal in its length.

5.2 Nested Queries

Handling arbitrary nested queries is a hard task on its own. In our current prototype, we only handle a subset of possible SQL nestings by adding additional templates that represent common forms of nested queries where the slots for the outer and inner query can be instantiated individually. An example for a NL-SQL template pair looks as follows:

SQL Template:
Select {Attribute}(s) From {Table} Where (Select {MaxMinAttribute} From {Table} Where {Filter}))
NL Template:
{SelectPhrase} the {Attribute}(s) {FromPhrase} {Table} {WherePhrase} {MaxMinAttribute}

This template is then instantiated during the first phase of the data generation process. For example, the following pair of instantiated queries could be generated for the training set from the previous template pair:

SQL Query (Training Set):
`SELECT name FROM mountain WHERE height = (SELECT MAX(height) FROM mountain WHERE state=@STATE.NAME)`
NL Query (Training Set):
What is name of the mountain with maximum height in @STATE.NAME

The instantiated queries are augmented automatically in the same way as for non-nested queries. In its current version, DBPal only supports uncorrelated nestings in the `WHERE` clause using different keywords (e.g., `EXISTS`, `IN`), as well as nested queries where the inner query returns an aggregate result. However, the nesting capabilities of DBPal can easily be extended by further adding templates that are instantiated in the first phase of the data generation.

6 EXPERIMENTAL EVALUATION

The main goal of our evaluation is to show that the presented training pipeline is able to improve the performance of existing NL2SQL translation techniques. Therefore, in Section 6.1, we first compare our proposed augmentation techniques to the training process using SyntaxSQLNet [46] with the well-known Spider [47] benchmark. Based on this analysis, in Section 6.2, we introduce a new benchmark for NLIDBs that better tests linguistic variations for NL2SQL translation and present experimental results. Finally, Section 6.3 presents the results of several microbenchmarks that test different aspects of DBPal's training pipeline.

6.1 Existing Benchmark: Spider

The first benchmark that we use to show the effectiveness of our proposed techniques is Spider [47]. In the following, we describe the benchmark at a high-level, and then we show how DBPal can effectively improve the accuracy of SyntaxSQLNet [46] on the Spider benchmark. SyntaxSQLNet is a state-of-the-art deep learning model that uses pre-trained GloVe word embeddings [30] when parsing the words in the input sentences. Using GloVe embeddings already allows the model to handle variations of individual words efficiently.

6.1.1 Setup. Spider [47] is a popular openly-available dataset that consists of over 10,000 NL questions paired with the corresponding SQL queries. The benchmark contains 200 database schemas, each of which has several tables, representing real-world database deployments. The data in the benchmark is very diverse and spans 138 distinct domains (e.g., automotive, social networking, geography). In addition to the diverse data, the corresponding SQL queries contain almost all of the common SQL patterns, including nested queries.

Algorithm	Easy	Medium	Hard	Very Hard	Overall
SyntaxSQLNet	0.445	0.227	0.231	0.051	0.248
DBPal (Train)	0.472	0.300	0.252	0.107	0.299
DBPal (Full)	0.480	0.323	0.279	0.122	0.317

Table 2: Spider Benchmark Results

Based on the complexity of the corresponding SQL query (i.e., the number of SQL components), each question is assigned a difficulty (i.e., easy, medium, hard, very hard). The benchmark includes queries from each of these categories, allowing us to test how different approaches compare in a diverse set of scenarios. In this benchmark, accuracy is measured by computing the number of correctly translated NL phrases divided by the total number of queries. A query is deemed to be correctly translated only if it exactly matches the provided "gold standard" SQL query for the NL input, without allowing for semantically equivalent answers.

Unlike existing datasets, Spider uses different databases (i.e., schemas and data) for training and testing (i.e., a database schema is used exclusively for either training or testing, but not both). This allows us to evaluate how well the model will generalize to new domains.

6.1.2 Results. Table 2 shows the accuracy for SyntaxSQLNet using the Spider dataset for three different configurations. First, as a baseline, we show the performance of the base SyntaxSQLNet model trained using the data from Spider's training set. The DBPal (Train) configuration uses the baseline SyntaxSQLNet (i.e., trained using Spider's training set), but we augmented the training data with additional synthetic data generated by DBPal using the schemas of the training set in Spider only. Finally, the DBPal (Full) version uses the schemas from both the training and test set of Spider to generate additional synthetic training data. Note, however, that DBPal never sees actual NL-SQL pairs from the test set during the training process, only the schemas in the DBPal (Full) configuration.

As shown, both configurations of DBPal improve upon the baseline performance of SyntaxSQLNet across all difficulty levels. In the DBPal (Train) case, we see that with the addition of synthetic training data generated only using schema information from the training set, DBPal is already able to outperform the baseline SyntaxSQLNet model. This is due to the fact that our novel training pipeline is able to supplement the existing training data with additional query patterns (e.g., nested subqueries) that are not present (or numerous enough) in the training data. As shown, this helps significantly for the harder queries, with DBPal being able to outperform the baseline by more than 2× for the "very hard" category due to the fact that the training pipeline introduces new query patterns (e.g., nested queries) to the model.

In general, DBPal (Full) is able to leverage additional query patterns from the synthetic data generation pipeline that are specific for the test schemas. With this information, DBPal (Full) is able to generate training examples that provide the model with additional information (e.g., table names, column names, column values) that is specific to test databases. As shown in Table 2, the added synthetic data for the test schemas in Spider when using DBPal (Full) is able to offer additional performance improvement over DBPal (Train). More concretely, with the help of the additional generated training data, we can further improve translation accuracy across all query difficulties of Spider by 15 − 27%.

6.2 New Benchmark: Patients

While Spider covers both a wide variety of schemas from different domains and different SQL query patterns, it does not comprehensively test different linguistic variations. Hence, we introduced a new open-source NL2SQL benchmark[1] that is available online specifically to test a model's linguistic robustness.

6.2.1 Setup. The schema of our new benchmark models a medical database comprised of hospital patients with attributes such as name, age, and disease. We refer to this dataset as the Patients benchmark. In total, the benchmark consists of 399 carefully crafted pairs of NL-SQL queries.

To better test the linguistic robustness of the given translation model, queries are grouped into one of the following categories depending on the linguistic variation that is used in the NL query: naive, syntactic paraphrases, morphological paraphrases, semantic paraphrases, and lexical paraphrases, as well as a category where queries have some missing information. These categories are formulated along the guidelines of paraphrase typologies discussed in [41] and [6]. While the NL queries in the naive category represent a direct translation of their SQL counterpart, the other categories are more challenging: syntactic paraphrases emphasize structural variances, lexical paraphrases pose challenges such as synonymous words and phrases, semantic paraphrases use changes in lexicalization patterns that maintain the same semantic meaning, morphological paraphrases add affixes, apply stemming, etc., and the missing category includes implicit references to concepts.

[1] https://github.com/DataManagementLab/ParaphraseBench

Algorithm	Naive	Syntactic	Lexical	Morphological	Semantic	Missing	Mixed	Overall
SyntaxSQLNet	0.281	0.228	0.070	0.175	0.175	0.088	0.140	0.165
DBPal (Train)	0.930	0.333	0.404	0.667	0.228	0.088	0.193	0.409
DBPal (Full)	0.947	0.632	0.544	0.667	0.491	0.158	0.298	0.531

Table 3: Patients Benchmark Results

Unlike other benchmarks that test for exact syntactic match of SQL queries, Patients tests instead for semantic equivalence. Since the test set is (relatively) small (i.e., 57 queries per category), we manually enumerated possible semantically equivalent SQL query answers. However, if the benchmark were to be extended, one could use an equivalence checker (e.g., Cosette [9]) to verify correctness.

In the following, we show an example query for each of these categories:

Naive: *"What is the average length of stay of patients where age is 80?"*

Syntactic: *"Where age is 80, what is the average length of stay of patients?"*

Morphological: *"What is the averaged length of stay of patients where age equaled 80?"*

Lexical: *"What is the mean length of stay of patients where age is 80 years?"*

Semantic: *"On average, how long do patients with an age of 80 stay?"*

Missing Information: *"What is the average stay of patients who are 80?"*

6.2.2 Results. In this section, we show how our proposed techniques compare using the previously described Patients benchmark. Table 3 shows the performance of SyntaxSQLNet (Baseline), our proposed synthetic data generation using only information from the training set (DBPal (Train)), and synthetic data generation using schema information from both the training and testing set (DBPal (Full)).

In the results, we see that our proposed synthetic data generation techniques can help improve the performance of SyntaxSQLNet across all of the linguistic variation categories. In particular, our techniques improve the translation accuracy by almost 25% by generating additional training data over only the training set and can provide a more than 35% accuracy improvement over SyntaxSQLNet by leveraging schema information about the test databases.

In general, the results of our training data augmentation fall into two categories. On one hand, there are query pattern categories where the baseline DBPal augmentation achieves almost all of the observed performance improvement (e.g., Naive, Morphological). In these cases, DBPal improves model performance by providing training examples for classes of queries that are not well-covered by the Spider training set, and the target schema knowledge provides virtually no additional benefit.

The second category of query patterns is where there is a large performance difference between DBPal (Train) and the target schema augmentation version, DBPal (Full), where accuracy is often doubled (e.g., semantic, missing). In these categories, the additional schema information is particularly helpful because it allows the model to learn complex, domain-specific NL mappings. For example, consider the example semantic query: "On average, how long do patients older than 80 stay?" Clearly, the semantic meaning of the phrase "older than" refers implicitly to the "age" attribute of the patient, but this would not be easy to derive from a generic training set. However, by providing training data specifically generated from the target schema, DBPal is able to help the model to better learn these mappings.

6.3 Microbenchmarks

In the following, we present the results of our microbenchmarks, which include: (1) an analysis of Spider results based on SQL pattern coverage; (2) the sensitivity of DBPal when using only a fraction of seed templates; and (3) our hyperparameter optimization techniques described in Section 3.3.

6.3.1 Pattern Coverage Breakdown. To understand the specific benefits of DBPal, we analyzed our results for the Spider benchmark from Section 6.1 based on query pattern coverage in the training data. Table 4 shows the same overall performance results reported in Table 2 broken down by query patterns in the test set using the following categories: the query pattern of the test query was found in (1) both the Spider training set and augmented data generated by DBPal; (2) only the augmented DBPal data; (3) only the training set of Spider; and (4) neither of them. For example, the simple `SELECT COUNT(*)` query pattern appears in both training sets, whereas only the Spider training dataset has coverage for multiple nested subqueries.

In general, we see that DBPal improves accuracy for all four categories, demonstrating that our data augmentation process can improve linguistic robustness irrespective of which training set contains individual query patterns. That is, DBPal's generated data actually helps the model to generalize and be more linguistically robust to patterns that are not explicitly covered in our seed templates. This effect can

Algorithm	Both	DBPal	Spider	Unseen
SyntaxSQLNet	0.375	0.000	0.244	0.013
DBPal (Train)	0.458	0.000	0.287	0.026
DBPal (Full)	0.462	0.250	0.317	0.040

Table 4: Pattern Coverage Breakdown for Spider

be seen for patterns that appear only in the Spider training dataset (*Spider*), where DBPal improves the model performance by about 30%. Even more impressive is the over 3× improvement for test queries where the query patterns never explicitly appear in any training set (*Unseen*).

Again, as observed in our other results, the additional augmentation step using the target schema further increases accuracy. For the *Both* category, this enables model specialization of those patterns to the target schema, whereas for the *Spider* and *Unseen* categories, it helps the model to learn to translate patterns with no DBPal coverage to the target schema.

Finally, one notable result is for the query patterns that appear only in DBPal's seed templates. As expected, the baseline SyntaxSQLNet model has never seen these query patterns (since they do not appear in Spider) and thus has 0% accuracy, whereas DBPal achieves 25% accuracy by learning from augmented examples of these patterns.

6.3.2 Seed Templates. Since DBPal generates additional training data by instantiating seed templates, the number of templates used during training can impact the overall benefit of our training pipeline. Therefore, to demonstrate the impact of the seed templates, Figure 3 shows the normalized accuracy (i.e., performance compared to using all of the templates) using the Patients benchmark when varying the number of templates used during training.

For this experiment, we train the same SyntaxSQLNet model using the previously mentioned Spider training data and include additional training data that is generated for the Patients schema only using a random subset of the available seed templates. For example, in the 10% case, we augment the Spider data with additional training examples that are instantiated using a randomly selected 10% of the available seed templates on the Patients schema. Importantly, the random subsets are selected prior to instantiation, which means templates covering certain patterns are excluded.

As shown, the addition of only 10% of the available seed templates is able to improve the overall accuracy when running the Patients test queries by more than 4×. Adding even more of the available seed templates (i.e., 50%) offers an additional 15% accuracy improvement, showing that additional templates are able to capture distinct NL2SQL patterns.

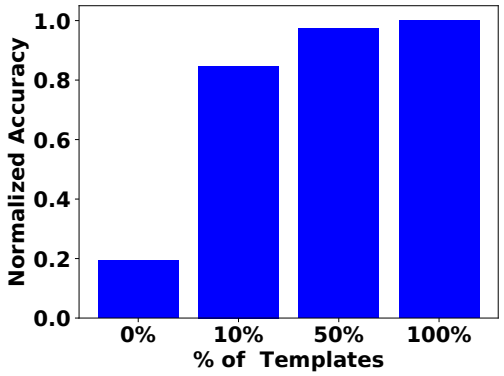

Figure 3: Normalized Accuracy for Fractions of Seed Templates

6.3.3 Hyperparameter Optimization. As described in Section 3.3, we apply an automatic hyperparameter optimization procedure to tune the parameters of our training data generator. In this experiment, we show the results of applying our optimization procedure for generating the training data for the Spider benchmark we used in our experiments in Section 6.1.

As a test workload T to tune the hyperparameters of our data generation pipeline, we used the full GeoQuery query test set of 280 pairs provided by [22]. The rationale is that the GeoQuery queries are partially included in the Spider test set and thus represents a good test set for the hyperparameter tuning, since the queries can be seen as representative on the one hand but also independent from the actual Spider test set. For the experiment, we sampled 68 random sets of hyperparameters. For every set of randomly sampled hyperparameters, we then trained a given model for up to a 6 hour time limit (which we saw is the typical time a model needs to converge when trained on Spider and DBPal training data).

Figure 4 shows the distribution of the accuracy recorded from running the optimization procedure, which trains a model on every dataset that was generated using the randomly sampled hyperparameters. Of the 68 parameter sets we evaluated, 59 converged within their 6 hour time limit. The worst model returned had an accuracy of 37.5%, while the best had an accuracy of 55.5%. The mean accuracy of all 59 models was 48.4%, with a standard deviation of 3.5%.

We used the hyperparameters which returned the highest accuracy as the basis for all other experiments previously described in this section.

7 RELATED WORK

The task of synthesizing SQL queries from natural language (NL) has been studied extensively within both the NLP and database research communities since the late 1970s [31, 48].

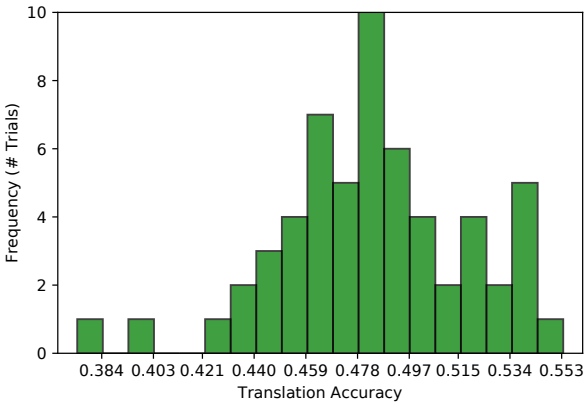

Figure 4: Histogram of Test Accuracy for Enumerated Parameter Configurations

A 1995 study [2] extensively discusses challenges that need to be addressed pertaining to Natural Language Interfaces for Databases (NLIDBs); their list includes linguistic coverage and database portability.

NLIDBs have a long history in the database research community [2, 25, 26, 32, 33, 36]. Most of this work relied on classical techniques for semantic parsing and used rule-based approaches for the translation into SQL. However, these approaches have commonly shown poor flexibility for the users who phrase questions with different linguistic styles using paraphrases and thus failed to support realistic scenarios.

More recent approaches tackled some of the limitations of the original NLIDBs. For example, the system ATHENA [36] relies on a manually crafted ontology that is used to make query translation more robust by taking different ways of phrasing a query into account. Yet, since ontologies are domain-specific, they need to be hand-crafted for every new database schema. On the other hand, the NaLIR [25] system relies on an off-the-shelf dependency parser that could also be built on top of a deep model. However, it still implements a rule-based system that struggles with variations in vocabulary and syntax. Our system attempts to solve both of those issues by being domain-independent as well as robust to linguistic variations.

Within the NLP community, this task is most commonly treated as a semantic parsing problem where the goal is to model a mapping of NL to a corresponding logical form, in this case SQL. Earlier works, such as [3, 4, 27, 49], employ variants of CCG parsers [10] to parse NL utterances into an executable lambda calculus notation. It should be noted that the grammar of logical form notation is far more simplistic than that of a complex query language like SQL; as such, a single NL query can be mapped to an arbitrarily complex SQL statement crossing many tables and involving many layers of nesting.

Recent success in employing neural network sequence-to-sequence (seq2seq) modeling for syntactic constituency parsing by [42] has spurred efforts in adapting the same solution for semantic parsing. That is, they pose logical form synthesis as a neural machine translation task, adapting systems for translating English to Czech or French to instead treat the logical form as the target foreign language. In both settings, mapping to lambda calculus [16, 17] or directly to SQL [8, 19, 22], the seq2seq architecture has shown competitive performance with statistical approaches that rely heavily upon hand-crafted lexical features.

In general, seq2seq models consist of a large number of parameters that require vast amounts of training examples. This poses a substantial challenge, as collecting diverse enough training data comprising pairs of NL utterances and logical form or SQL queries requires expensive expert supervision. Iyer et al. [22] attempts to deal with this data bottleneck by performing an online learning mechanism in which the model alternates between training and making predictions. Human judges identify incorrect predictions that need to be corrected by a crowdsourced worker with SQL expertise.

Alternatively, a solution more similar to ours is introduced by [43], whose approach produces pairs of canonical utterances aligned with their corresponding logical forms using a seed lexicon. However, they again use crowdsourcing to paraphrase the canonical utterances into more fluent sentences that include syntactic alterations and context specific predicates. While less efficient than an on-the-fly system, this form of crowdsourced annotation is much less costly, given worker's SQL expertise is not required.

The main contribution of this work addresses the training data bottleneck from a slightly different angle. We attempt to completely eliminate any manual annotation effort by a user who is not well-versed in SQL. Rather, the user needs to be familiar only with the given new domain in order to sufficiently annotate the new schema's elements with their NL utterances. We argue that our extensive linguistically-aware templates provide a comparable breadth of coverage as that of manually collected training data. Additionally, our strategy of employing PPDB [29] to automatically paraphrase the sentence can approximate a human doing the same task.

Previous work on Natural Language Processing (NLP) has heavily relied on classical statistical models to implement tasks such as semantic parsing that aim to map a natural language utterance to an unambiguous and executable logical form [48]. More recent results on semantic parsing such as [16, 23] have started to employ deep recurrent neural networks (RNNs), particularly seq2seq architectures, to replace traditional statistical models. RNNs have shown promising results and outperform the classical approaches for semantic parsing, since they make only few domain-specific assumptions and thus require only minimal feature engineering.

An important research area aiming to allow non-experts to specify ad-hoc queries over relational databases are keyword search interfaces [45]. Recently, there have been extensions to keyword-based search interfaces to interpret the query intent behind the keywords in the view of more complex query semantics [5, 7, 39]. In particular, some of them support aggregation functions, boolean predicates, etc.

Some recent approaches leverage deep models for end-to-end translation similar to our system (e.g., [22]). However, a main difference of our system to [22] is that their approach requires manually handcrafting a training set for each novel schema/domain that consist of pairs of NL and SQL queries. In contrast, our approach does not require a hand-crafted training set. Instead, inspired by [43], our system generates a synthetic training set that requires only minimal annotations to the database schema.

Another recent paper that also uses a deep model to translate NL to SQL is [44]. First, the approach in this paper is a more classical approach based on identifying the query intent and then filling particular slots of a query. In their current version [44], they can only handle a much more limited set of NL queries compared to DBPal. Furthermore, their approach leverages reinforcement learning to learn from user feedback in case the query could not be translated correctly, which is an orthogonal issue that could also be applied to DBPal.

Finally, in addition to its primary focus on generating labels for unlabeled training data, Snorkel [34] also incorporates data augmentation techniques to generate additional heuristically modified training examples [14, 35]. Unlike Snorkel, DBPal presents many optimizations that are specific to the task of NL2SQL translation, including slot-fill dictionaries, random word-dropout, and paraphrasing techniques to increase the linguistic robustness of the generated training data. Additionally, DBPal includes an optimization procedure for hyper-parameter tuning that leverages schema information to further specialize the generated training examples for the target use case.

8 CONCLUSION & FUTURE WORK

In this paper, we presented DBPal, a fully pluggable natural language to SQL (NL2SQL) training pipeline that generates synthetic training data to improve both the accuracy and robustness to linguistic variation of existing deep learning models. In combination with our presented data augmentation techniques, which help improve the translational robustness of the underlying models, DBPal is able to improve the accuracy of state-of-the-art deep learning models by up to almost 40%.

Longer term, we believe that an exciting opportunity exists to expand DBPal's techniques to tackle broader data science use cases, ultimately allowing domain experts to interactively explore large datasets using only natural language [24]. In contrast to the typical notion of one-shot SQL queries currently taken by DBPal, data science is an iterative, session-driven process where a user repeatedly modifies a query or machine learning model after examining intermediate results until finally arriving at some desired insight, which will therefore necessitate a more conversational interface. These extensions would require the development of new techniques for providing progressive results [40, 50] by extending past work on traditional SQL-style queries [13, 20] and machine learning models [37].

Finally, we believe there are also interesting opportunities related to different data models (e.g., time series [18]) and new user interfaces (e.g., query-by-voice [28]).

9 ACKNOWLEDGMENTS

This work was funded in part by NSF grants III:1526639 and III:1514491, as well as gifts from Oracle to support our work on Natural Language Interfaces on Big Data.

REFERENCES

[1] 2020. Tableau. https://www.tableau.com/. (2020).
[2] Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. 1995. Natural language interfaces to databases - an introduction. *Natural Language Engineering* 1, 1 (1995), 29–81.
[3] Islam Beltagy, Katrin Erk, and Raymond Mooney. 2014. Semantic Parsing using Distributional Semantics and Probabilistic Logic. In *ACL 2014 Workshop on Semantic Parsing*. 7–11.
[4] Jonathan Berant and Percy Liang. 2014. Semantic Parsing via Paraphrasing. In *ACL*. 1415–1425.
[5] Sonia Bergamaschi, Francesco Guerra, Matteo Interlandi, Raquel Trillo Lado, and Yannis Velegrakis. 2013. QUEST: A Keyword Search System for Relational Data based on Semantic and Machine Learning Techniques. *PVLDB* 6, 12 (2013), 1222–1225.
[6] Rahul Bhagat and Eduard H. Hovy. 2013. What Is a Paraphrase? *Computational Linguistics* 39, 3 (2013), 463–472.
[7] Lukas Blunschi, Claudio Jossen, Donald Kossmann, Magdalini Mori, and Kurt Stockinger. 2012. SODA: Generating SQL for Business Users. *PVLDB* 5, 10 (2012), 932–943.
[8] Ruichu Cai, Boyan Xu, Zhenjie Zhang, Xiaoyan Yang, Zijian Li, and Zhihao Liang. 2018. An Encoder-Decoder Framework Translating Natural Language to Database Queries. In *IJCAI*. 3977–3983.
[9] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *CIDR*.
[10] Stephen Clark and James R. Curran. 2004. Parsing the WSJ Using CCG and Log-Linear Models. In *ACL*. 103–110.
[11] Mark Craven, Dan DiPasquo, Dayne Freitag, Andrew McCallum, Tom M. Mitchell, Kamal Nigam, and Seán Slattery. 2000. Learning to construct knowledge bases from the World Wide Web. *Artif. Intell.* 118, 1-2 (2000), 69–113.
[12] Andrew Crotty, Alex Galakatos, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2015. Vizdom: Interactive Analytics through Pen and Touch. *PVLDB* 8, 12 (2015), 2024–2027.
[13] Andrew Crotty, Alex Galakatos, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2016. The case for interactive data exploration accelerators (IDEAs). In *HILDA@SIGMOD*.

[14] Tri Dao, Albert Gu, Alexander Ratner, Virginia Smith, Chris De Sa, and Christopher Ré. 2019. A Kernel Theory of Modern Data Augmentation. In *ICML*, Vol. 97. 1528–1537.

[15] Mostafa Dehghani, Hamed Zamani, Aliaksei Severyn, Jaap Kamps, and W. Bruce Croft. 2017. Neural Ranking Models with Weak Supervision. In *SIGIR*. 65–74.

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

[17] Li Dong and Mirella Lapata. 2018. Coarse-to-Fine Decoding for Neural Semantic Parsing. In *ACL*. 731–742.

[18] Philipp Eichmann, Andrew Crotty, Alexander Galakatos, and Emanuel Zgraggen. 2017. Discrete Time Specifications In Temporal Queries. In *CHI Extended Abstracts*. 2536–2542.

[19] Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir R. Radev. 2018. Improving Text-to-SQL Evaluation Methodology. In *ACL*. 351–360.

[20] Alex Galakatos, Andrew Crotty, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2017. Revisiting Reuse for Approximate Query Processing. *PVLDB* 10, 10 (2017), 1142–1153.

[21] Mi-Young Huh, Pulkit Agrawal, and Alexei A. Efros. 2016. What makes ImageNet good for transfer learning? *CoRR* abs/1608.08614 (2016).

[22] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a Neural Semantic Parser from User Feedback. In *ACL*. 963–973.

[23] Robin Jia and Percy Liang. 2016. Data Recombination for Neural Semantic Parsing. In *ACL*.

[24] Rogers Jeffrey Leo John, Navneet Potti, and Jignesh M. Patel. 2017. Ava: From Data to Insights Through Conversations. In *CIDR*.

[25] Fei Li and H. V. Jagadish. 2014. Constructing an Interactive Natural Language Interface for Relational Databases. *PVLDB* 8, 1 (2014), 73–84.

[26] Fei Li and Hosagrahar Visvesvaraya Jagadish. 2014. NaLIR: an interactive natural language interface for querying relational databases. In *SIGMOD*. 709–712.

[27] Percy Liang, Michael I. Jordan, and Dan Klein. 2011. Learning Dependency-Based Compositional Semantics. In *ACL*. 590–599.

[28] Gabriel Lyons, Vinh Tran, Carsten Binnig, Ugur Çetintemel, and Tim Kraska. 2016. Making the Case for Query-by-Voice with EchoQuery. In *SIGMOD*. 2129–2132.

[29] Ellie Pavlick and Chris Callison-Burch. 2016. Simple PPDB: A Paraphrase Database for Simplification. In *ACL*.

[30] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global Vectors for Word Representation. In *EMNLP*. 1532–1543.

[31] 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 *COLING*.

[32] Ana-Maria Popescu, Oren Etzioni, and Henry A. Kautz. 2003. Towards a theory of natural language interfaces to databases. In *IUI*. 149–157.

[33] Rodolfo A. Pazos Rangel, Joaquín Pérez Ortega, Juan Javier González Barbosa, Alexander F. Gelbukh, Grigori Sidorov, and Myriam J. Rodríguez M. 2005. A Domain Independent Natural Language Interface to Databases Capable of Processing Complex Queries. In *MICAI*, Vol. 3789. 833–842.

[34] Alexander Ratner, Stephen H. Bach, Henry R. Ehrenberg, Jason Alan Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: Rapid Training Data Creation with Weak Supervision. *PVLDB* 11, 3 (2017), 269–282.

[35] Alexander J. Ratner, Henry R. Ehrenberg, Zeshan Hussain, Jared Dunnmon, and Christopher Ré. 2017. Learning to Compose Domain-Specific Transformations for Data Augmentation. In *NIPS*. 3236–3246.

[36] Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R. Mittal, and Fatma Özcan. 2016. ATHENA: An Ontology-Driven System for Natural Language Querying over Relational Data Stores. *PVLDB* 9, 12 (2016), 1209–1220.

[37] Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, and Tim Kraska. 2019. Democratizing Data Science through Interactive Curation of ML Pipelines. In *SIGMOD*. 1171–1188.

[38] Chen Sun, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta. 2017. Revisiting Unreasonable Effectiveness of Data in Deep Learning Era. In *ICCV*. 843–852.

[39] Sandeep Tata and Guy M. Lohman. 2008. SQAK: doing more with keywords. In *SIGMOD*. 889–902.

[40] Cagatay Turkay, Nicola Pezzotti, Carsten Binnig, Hendrik Strobelt, Barbara Hammer, Daniel A. Keim, Jean-Daniel Fekete, Themis Palpanas, Yunhai Wang, and Florin Rusu. 2018. Progressive Data Science: Potential and Challenges. *CoRR* abs/1812.08032 (2018).

[41] Marta Vila, Maria Antònia Martí, and Horacio Rodríguez. 2011. Paraphrase Concept and Typology. A Linguistically Based and Computationally Oriented Approach. *Procesamiento del Lenguaje Natural* 46 (2011), 83–90.

[42] Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey E. Hinton. 2015. Grammar as a Foreign Language. In *NIPS*. 2773–2781.

[43] Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a Semantic Parser Overnight. In *ACL*. 1332–1342.

[44] Xiaojun Xu, Chang Liu, and Dawn Song. 2017. SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning. *CoRR* abs/1711.04436 (2017).

[45] Jeffrey Xu Yu, Lu Qin, and Lijun Chang. 2010. Keyword Search in Relational Databases: A Survey. *IEEE Data Eng. Bull.* 33, 1 (2010), 67–78.

[46] Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir R. Radev. 2018. SyntaxSQLNet: Syntax Tree Networks for Complex and Cross-Domain Text-to-SQL Task. In *EMNLP*. 1653–1663.

[47] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *EMNLP*. 3911–3921.

[48] John M. Zelle and Raymond J. Mooney. 1996. Learning to Parse Database Queries Using Inductive Logic Programming. In *AAAI*. 1050–1055.

[49] Luke S. Zettlemoyer and Michael Collins. 2007. Online Learning of Relaxed CCG Grammars for Parsing to Logical Form. In *EMNLP*. 678–687.

[50] Emanuel Zgraggen, Alex Galakatos, Andrew Crotty, Jean-Daniel Fekete, and Tim Kraska. 2017. How Progressive Visualizations Affect Exploratory Analysis. *IEEE Trans. Vis. Comput. Graph.* 23, 8 (2017), 1977–1987.

[51] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *CoRR* abs/1709.00103 (2017).