Synthesis of SQL Queries from Natural Language Descriptions

AALOK THAKKAR, University of Pennsylvania, USA

A natural language interface to a database is a system that allows an end-user to query a database using natural language expressions. Since 1980s, the construction of such interfaces for databases has attracted attention from both the database and natural language communities. This report addresses the problem of synthesizing relational queries, in a structured querying language like SQL, from natural language descriptions, which forms the core of these interfaces. We begin with a brief overview of the problem and its history, and then focus on three recent state-of-the art techniques: Neural Refinement [6], Type-driven Refinement [24], and Dual Specifications [1]. We conclude with a discussion of their strengths and limitations.

Presentation Date: February 7, 2022.
Committee: Lyle Ungar, Susan Davidson, Alex Polozov
Intended Audience: First year graduate students at the department of Computer and Information Science.

1 INTRODUCTION

Accessing a database requires some expertise. Prior to 1970, programmers would need to understand the structure of the stored data and learn programming techniques in order to query the database. With the introduction to the relational model of data [5], databases became more accessible as the only storage structure was a table, something that even naive users could understand. This led to the development of declarative relational query languages such as SQL simplifying database programming.

Consider a relational database as shown in Figure 2. The first table, Papers, shows the name of the paper published, the conference it is published in, and the year of publication. The second table, Conferences, shows the name of the conference, year of the conference, and venue. The third table, Cities, maps cities to countries. The Output Table, mapping the paper to the country it was presented in can be generated using the SQL query Q_0:

```
SELECT Papers.Name, Cities.Country
    FROM Papers
    JOIN Conferences
        ON (Papers.Conference = Conferences.Name)
        AND (Papers.Year = Conferences.Year)
    JOIN Cities
        ON (Conferences.Venue = Cities.Name);
```

Fig. 1. SQL query Q_0 mapping research papers to the country it was presented in.

The SQL query above instructs the database system to report all possible pairs P, Y consisting of a paper name P and country Y such that there exists a conference C, year Y, and city X such that (P, C, Y) is a row in Papers table, (C, Y, X) is a row in Conferences table, and (X, Y) is a row in Cities table.

SQL query is a high-level logical specification and SQL query processor takes care of how to efficiently implement this query to produce the desired output table, still such queries can be difficult to write, particularly when there

Author's address: Aalok Thakkar, University of Pennsylvania, USA, athakkar@cis.upenn.edu.

Papers

Name	Conference	Year
Coarse-to-Fine Decoding for Neural Semantic Parsing	ACL	2018
SQLizer query synthesis from natural language	OOPSLA	2017
Duoquest: A Dual-Specification System for Expressive SQL Queries	PODS	2020
...	...	...

Conferences

Name	Year	Venue
OOPSLA	2017	Vancouver
OOPSLA	2018	Boston
ACL	2018	Melbourne
ACL	2019	Florence
PODS	2020	Oregon
...	...	...

Cities

Name	Country
Boston	USA
Melbourne	Australia
Oregon	USA
Philadelphia	USA
Vancouver	Canada
...	...

Output Table

Name	Country
Coarse-to-Fine Decoding for Neural Semantic Parsing	Australia
SQLizer query synthesis from natural language	Canada
Duoquest: A Dual-Specification System for Expressive SQL Queries	USA
...	..

Fig. 2. Tables representing the input relational databases for Papers, Conferences, and Cities, and output database for research papers mapped to the country they were presented in. The query Q_0 maps the input to the given output.

are multiple tables and attributes involved. Additionally, one may use SQL's highly expressive constructs such as aggregations, correlated sub-queries, unions, nested queries, groupings, pivoting, various types of joins, etc. to form advance idioms. This makes mastering SQL a challenge. As a result, there has been a great interest in automatically synthesizing relational queries from intuitive specifications.

An intuitive specification corresponding to query Q_0 in English can be:

"Find country of publication for each paper in Papers table"

This report focuses on techniques that translate such natural language descriptions to relational queries. We present an overview and history of the problem of synthesising relational queries from natural language descriptions in Section 2, and then survey three techniques that advance the state-of-the-art for NL-based synthesis:

(1) In Section 3, we look at a neural architecture that decomposes the synthesis problem into two phases: generating a sketch of the query without low-level information (such as variable names and arguments) and then filling in missing details [6],
(2) In Section 4, we discuss a refinement technique, that uses type-directed program synthesis and automated program repair to complete a query sketch, and

(3) In Section 5, we review the paradigm of Programming-by-Examples (PBE) for relational queries, and discuss Duoquest [1], a technique combining NL-based synthesis with PBE.

We conclude this report with a discussion of the strength and limitations of these three approaches.

2 OVERVIEW

The construction of natural language interfaces for databases has been studied in both the database and natural language communities for decades. We start with a brief history of the problem, discuss the advantages and limitations of the approach, and end this section with the formulation of the problem.

2.1 A Brief History

Natural language interfaces for databases started appearing in 1960s, and Lunar [22], a natural language interface to a database containing chemical analyses of moon rock, is the one of the popular interface of that period. Lunar was built with a particular database and application in mind and had limited portability.

In the 1980s, methods such as Ladder and Chat-80 that use intermediate logical representations were proposed. Ladder operated independent of the underlying database schema, but used domain-specific hand-crafted mapping rules for translation. Chat-80 implemented a question answering system for world geography, that translates the given English question into a Prolog expression and then evaluates it against a Prolog database. Most advances in these times and the following decade, such as Team, Ask, and Janus focused on portability issues.

From the early 2000s, more advanced rule-based methods were proposed. Precise [14] uses statistical parsing techniques that associate the rules with probabilities. The technique requires that words in the natural language description have a one-to-one correspondence with the set of database elements. This limitation is overcome by techniques that associate a mapping score with potential candidates and rank them [9, 16, 24]. Although these methods achieved significant performance improvement, they still rely on manually-defined rules.

Recently, numerous deep-learning-based methods have been proposed [2, 6, 8]. Two popular benchmark suites, WikiSQL and Spider, have driven efforts in this domain. Seq2SQL [27], SQLNet [23],and STAMP [19] proposed a new deep-learning model specific to WikiSQL, and SyntaxSQLNet [25] specific to Spider.

2.2 Advantages and Limitations

Natural language based querying offers a number of advantages:

(1) **No Artificial Language**: Formal query languages are difficult to learn and master, at least by non-experts. Graphical interfaces and form-based interfaces are easier but may still have operators that are artificial to occasional end-users. Natural language overcomes the need for user to learn new formalisms.
(2) **Expressiveness and Succinctness**: Natural languages are argued to be more succinct for expressing negation or quantification, compared to formal languages (such as SQL). For example, the question "Which company supplies every country?" can be translated to the SQL query but requires complex constructions
(3) **Discourse:** While communicating, we often build up context and use references (anaphora) and incomplete sentences (elliptical sentences) to communicate. For example, given two queries, "Which company supplies to China?" followed by "What other countries do *they* supply to?", the term 'they' clearly refers to the answer in the previous question. Anaphora and elliptical sentences have been well studied in natural language processing and tools such as LOQUI, Parlance, and XCalibur can handle them.

A number of NL-based existing techniques also suffer from limitations that include:

(1) **Task-Specific**: Certain NL-based approaches train on specific databases and may require significant overhead in adopting to new domains and databases [15, 17, 20, 26]
(2) **User Dependent**: Certain techniques require interactive guidance from the user [10], making them only partially automated.
(3) **Low Accuracy**: Recent advances in techniques using deep learning have overcome the above stated limitations but the state-of-the-art accuracy on established benchmark suites is low. [25]

2.3 Problem Formulation

Given a natural language description q and background knowledge BK, the goal of natural language-based translation is to synthesize a query Q_q in a given relational querying language (like SQL) such that Q_q and q are *semantically equivalent*. While formal semantics of structured languages are well defined, the semantics of natural language descriptions are not well-formed. A number of references discuss the semantics of natural language queries [**?**].

The background knowledge BK usually depends on the approach. For example, rule-based synthesis tools consider grammatical rules as their background knowledge and use it to parse the sentence, while neural methods may use training data as background knowledge. In Section 5, we discuss an approach that uses input-output examples as background knowledge.

3 NEURAL REFINEMENT

Using sketches as intermediate representations is a popular approach in program synthesis [18, 19]. The idea is to split the synthesis problem into two steps: (1) translate the specification into a sketch (or an abstract query where low-level information such as variable names and arguments are uninitialized) and then (2) fill in the missing details. Dong and Lapata [6] propose a neural architecture to achieve the same, and evaluate their approach on benchmarks from domains including logical representations for Geo/ATIS, Python source code, and SQL queries from the WikiSQL benchmark suite. Our discussion is focused on their work for SQL queries

3.1 Sketch Generation

WikiSQL queries follow the format "`SELECT agg_op agg_col WHERE (cond_col cond_op cond) AND ...`" which is a subset of SQL syntax. `SELECT` identifies the column that is to be included in the results after applying the aggregation operator `agg_op` to column `agg_col`. The operator `agg_op` comes from the set {empty, `COUNT`, `MIN`, `MAX`, `SUM`, `AVG`} and `agg_col` comes from the set {=, <, >}. `WHERE` can have zero or multiple conditions, which means that column `cond_col` must satisfy the constraints expressed by the operator `cond_op` and the condition value `cond`. Here is an example of a WikiSQL query:

```
SELECT Paper.Name WHERE (Paper.Year > 2019) AND (Paper.Conference = PODS)
```

Fig. 3. SQL query Q_1 to find papers published at PODS after 2019.

Sketches for SQL queries are simply the (ordered) sequences of condition operators `cond_op` in `WHERE` clauses. For example, for Q_1 in Fig 3, sketch "`WHERE > AND =`" has two condition operators, namely > and =. Only this sequence of condition operators are predicted in the sketch generation phase. The attributes for the `WHERE` clause and the aggregate and column for the `SELECT` clause are concretized in the refinement phase.

3.2 The Algorithm

Let x denote the input natural language expression, y denote a relational query, and $p(y|x)$ denote the likelihood of the relational query y corresponding to the expression x. In order to estimate the conditional probability, $p(y|x)$, we decompose the computation into a two stage generation process:

$$p(y|x) = p(y|a, x).p(a|x) \tag{1}$$

Where a is a sketch corresponding to y as discussed in Section 3.1. The core of the algorithm is to train a model that maximizes the log likelihood of the generated relational query y given natural expression x. Equation 1 allows for a greedy search by first finding $\hat{a} = \text{argmax}_a p(a|x)$ and then computing $\hat{y} = \text{argmax}_y p(y|\hat{a}, x)$.

The natural language input x, the sketch a, and the relational query y are all encoded as vectors. We will denote $x = x_1 \cdots x_{|x|}$, $y = y_1 \cdots y_{|y|}$, and $a = a_1 \cdots a_{|a|}$. Let $y_{<t} = y_1 \cdots y_{t-1}$. We can factorize $p(y|a, x)$ as:

$$p(y|a, x) = \prod_{t=1}^{|y|} p(y_t | y_{<t}, a, x) \tag{2}$$

Now, we have the language to look at the algorithm. The algorithm is divided into the following three steps: (1) Table-aware input encoding, (2) Generating `SELECT` clause, and (3), Generating `WHERE` clause.

Table-aware Input Encoding. A bi-directional recurrent neural network with long short-term memory units (LSTM [7]) is used to encode the natural language input x into vector representations. The word x is mapped to $\{\mathbf{e}_t\}_{t=1}^{x}$ using bi-directional LSTM units. Then, each header name c_k in the columns in the input tables is encoded as $\mathbf{c}_k$ using bi-directional LSTM units. Say we have M header names. For each t, we use an attention mechanism towards table column vectors $\{\mathbf{c}_k\}_{k=1}^{M}$ to obtain the most relevant columns for $\mathbf{e}_t$. Let $\mathbf{c}_t^e$ summarize the relevant columns for $\mathbf{e}_t$. We feed the concatenated vectors $\{[\mathbf{c}_t, \mathbf{c}_t^e]\}_{t=1}^{|x|}$ into a bi-directional LSTM encoder, and generate new encoding vectors $\{\tilde{\mathbf{e}}_t\}_{t=1}^{|x|}$.

Generating `SELECT` clause. The vector $\tilde{\mathbf{e}}$ is fed into a softmax classifier to obtain the aggregation operator `agg_op`. To predict the column `agg_col`, we use the table column vectors $\{\mathbf{c}_k\}_{k=1}^{M}$. Let $\sigma(\cdot)$ be a scoring network that matches the input x to column vectors $\mathbf{c}_k$. If `agg_col` is the k^{th} column, its probability is computed by:

$$p(\texttt{agg\_col} = k|x) \propto \exp\{\sigma(\tilde{\mathbf{e}}, \mathbf{c}_k)\} \tag{3}$$

Generating `WHERE` clause. We use the sketches to generate the `WHERE` clause. In the training data, there are only 35 sketches, allowing to model this as a classification problem. Treating sketches as categories, one can estimate $p(a|x)$ as:

$$p(a|x) = \text{softmax}_a(\mathbf{W}_a \tilde{\mathbf{e}} + \mathbf{b}_a) \tag{4}$$

Where $\mathbf{W}_a$ and $\mathbf{b}_a$ are parameters. Then, one can use this classifier as a sketch-generator. Once we have a sketch, it determines the number of `WHERE` clauses and it operators `cond_op`. It remains to find the columns `cond_col` and their values `cond`.

We use another set of bi-direction LSTM units to predict the columns cond_col. Let $\{\mathbf{h}_t\}_{t=1}^{|y|}$ denote the LSTM hidden states. We will use an attention mechanism [11] to learn soft alignments. Consider $s_{t,k}$ as the attention score for step t with respect to hidden state k. Then, we can compute:

$$\tilde{\mathbf{e}}_t^d = \sum_{k=1}^{|x|} s_{t,k}\tilde{\mathbf{e}}_k \tag{5}$$

$$\mathbf{h}_t^{att} = \tanh\left(\mathbf{W}_1\mathbf{h}_t + \mathbf{W}_2\tilde{\mathbf{e}}_t^d\right) \tag{6}$$

The condition column cond_col can be selected from the table's headers similar to the selection of agg_col as in Equation 3. We compute $p(\texttt{cond\_col} = k|y_{<t}, x, a)$ but use a different set of parameters and compute the score as $\sigma(\mathbf{h}_t^{att}, \mathbf{c}_k)$.

The condition values cond are typically mentioned in the natural language input x. Suppose it occurs as a phrase $x_l \cdots x_r$. Then one can compute $p(\texttt{cond}_{y_t} = x_l \cdots x_r|y_{<t}, a, x)$ by using another scoring network $\sigma(\cdot)$ (trained with a different set of parameters).

Put together, this gives us a way to estimate the likelihood $p(y|x)$. As we have the factorization in Equation 2, we can compute the output one token at a time.

3.3 Evaluation

This technique is implemented as a tool and evaluated on a set of benchmarks from WikiSQL. The accuracy is measured in comparison to the gold standard meaning representations. The results of the evaluation study conclude that the sketch encoder is central to the synthesis engine. Compared with previous neural models that utilize syntax or grammatical information, this method performs competitively despite the use of relatively simple decoders. The simulation study shows that the most gain is obtained by the improved decoder of the WHERE clause.

The authors also note that the input encoder must be tableaware as the same natural language specification can lead to different queries depending on schemas of the input tables.

3.4 Discussion

This paper presented a refinement framework for generating programs from natural language descriptions. The scope of this paper expands beyond relational queries. For SQL queries, it generates sketches which lists the conditional operators used in the WHILE clauses and then predicts missing details. Experimental results show that this technique improves performance over multiple domains. The performance improvement (against the baseline) for WikiSQL benchmark is incremental.

A major shortcoming of this method, as presented, is that it treats the problem of generating sketches as classification. This limits the method to adapt to benchmarks where one may not have a small number of sketches. The paper uses an alternative method to generate sketches for other domains (such as for logic expressions and Python programs) which can be adapted to the domain of relational queries.

Secondly, in this work, the training data includes the natural language input, the relational query output and the sketch. Sketches are not usually available with benchmarks such as that from WikiSQL or Spider. One may have to carefully curate the sketches for the benchmarks, which is a cumbersome task.

Thirdly, there is an opportunity to explore alternative formulations of sketches. The current sketch syntax is minimalist and only captures the conditionals used in the WHERE clause. It does not provide any details about the SELECT clause, or the columns and constants that appear in the WHERE clauses. Techniques such as those discussed in Section 4 use alternative abstractions.

4 TYPE-DIRECTED REFINEMENT

We now look at a type-directed refinement technique that uses program-repair techniques for synthesizing SQL queries. Similar to neural refinement, this line of work builds on programming by sketching. This technique is implemented as the tool SQLizer.

4.1 Sketch Generation

The first step for SQLizer is to use semantic parsing to generate a sketch from the natural language description. In this context, the paper targets a fragment defined as extended relational algebra that supports standard features such as selection, projection, and table joins, as well as aggregation functions such as maximum, minimum, average, sum, and count. Formally, the syntax can be expressed as a context free grammar, but for the purpose of this report, we will interpret the sketch as a SQL query with table names and column names as unknowns. The translation from extended relational algebra to SQL is a syntactic rewrite.

A feature of SQLizer's semantic parser is the use of hints. In essence, the natural language description contains some information to suggest ways to fill in the names of tables and columns. Exploiting these hints allows the synthesis tool to work well without requiring any database-specific training. The parser embeddes these hints in the sketch.

The paper has implemented a semantic parser top of the Sempre framework [3], which is a toolkit for building semantic parsers. For linguistic processing, the pre-trained models of the Stanford CoreNLP [12] library are used. Given a natural language description N, the parser generates all possible query sketches S_i and assigns each S_i a score that indicates the likelihood that S_i is the intended interpretation of N. This score is calculated based on a set of approximately 40 pre-defined features that it inherits from the Sempre framework such as the number of grammar rules used in the derivation, the length of the matched input, etc.

4.2 Quantitative Type Inhabitation

The next step is sketch completetion. Given a sketch S, the goal is to find an expression e such that when e is used to complete S, the probability of generating a query that is close to user intent is high. For this purpose, SQLizer interprets this problem as quantitative type inhabitation.

As discussed earlier, a key idea is to use natural language hints embedded in the the sketch S along with any domain-specific background knowledge that can be used to associate a confidence score with each such expression. The following are used to assign a confidence score:

(1) Names of schema elements using natural language hints for each hole,
(2) Foreign and primary keys that link between data in two different database tables
(3) Database contents, specifically when assigning scores to queries with SELECT clauses.

These domain-specific heuristics inform a set of quantitative type inhabitation rules for relations and the parameters used for selection and projection. The key idea here is that each instantiation should be well-typed, that is, it should be consistent with the schema of the appropriate database. Then, given a well-typed instantiation, the rules allow us to assign a confidence score p with the instantiation. For example, if there is sketch with an unknown database, any table can be used as an instantiation. However, using the natural language hint one can determine the likelihood of a given table being correct, that is, the similarity between hint and the table name informs the assignment of the confidence score. The other inhabitation rules allow us to compute the confidence score by composing the scores of smaller terms.

4.3 Sketch Refinement Using Repair

If quantitative type inhabitation procedure produces an inhabitation with high confidence score, we are done, and can return the corresponding query as an answer. However, in some cases, it may not be possible to find a well-typed inhabitation with high confidence. Two such cases are:

(1) Natural language is ambiguous and it is possible that the sketch for the intended query is not in the top k sketches generated by semantic parsing, and
(2) User's natural language description may be misleading as they may not be aware about the underlying data organization.

SQLizer overcomes them by using automated sketch refinement. Given a sketch S, the goal of sketch refinement is to generate a new sketch S' such that S' repairs a potentially faulty sub-part of S. This is achieved through a combination of fault localization and a database of repair tactics.

The fault localization is achieved by identifying a minimal fault F of S such that– (1) all inhabitants of F have a low threshold, and (2) all proper sub-expressions of F have an inhabitant with score at least greater than F. In order to find such an F, we start with the sketch S. If S is a relation, we recurse down to its subrelations and specifiers to identify a smaller subterm that can be repaired. On the other hand, if S is a specifier, we then recurse down to its subspecifiers, again to identify a smaller problematic subterm. If we cannot identify any such subterm, we then consider the current partial sketch S as the possible cause of failure. If so, we return S as the fault F. Once F is identified, then new predicates, join operators, or columns are introduced into F in order to repair it. The choice of these repairs is domain-specific.

4.4 Evaluation

SQLizer is evaluted on three different databases– MAS, IMDB, and YELP. It achieves about 90% accuracy across all three when we consider a benchmark to be successful if the desired query appears within the top 5 results. Even the definition of of success is to be made stricter to have the target query ranked as the top result, we still have about 78% accuracy. SQLizer takes about 1.2 seconds to synthesize a query and 85% of the runtime is spent in semantic parsing.

The simulation study also reveals that type information and sketch repair are central to the synthesis procedure.

4.5 Discussion

This paper presents a novel perspective on synthesis of relational queries. The experimental evaluation is wide and shows that SQLizer performs well on three sets of benchmarks.

A number of heuristics used in the paper are domain-specific and the evaluation reveals that they play an important role in the performance. While some of them such as seeking hints in the natural language descriptions are easily generalisable, it is unclear if one can learn such heuristics for different domains relatively easily.

It is exciting to witness the use of standard techniques from programming languages research being used in this domain and believe that similar ideas can be used for program synthesis beyond relational queries.

Summary of Programming-by-Example Techniques			
System	Expressiveness	Schema Knowledge	Output Table
QBE	SQL	Required	Partial
MWeaver	Only joins	Not requited	Exhaustive
SQuID	SQL without projected aggregates	Not required	Exhaustive
TALOS	SQL	Not required	Exhaustive
PALEO	Only selection and projection	Required	Exhaustive
Scythe	SQL	Required	Exhaustive
REGAL+	SQL	Not required	Exhaustive

Fig. 4. State-of-the art PBE techniques for synthesis of SQL queries are usually limited in expressiveness, require schema knowledge, or need exhaustive output tables. Highly expressive queries often require several rows in input and output tables to capture user intent.

5 A DUAL-SPECIFICATION

Natural language descriptions are often ambiguous, even for human interpreters, and using them for query synthesis is a challenge as one cannot verify if a given query is semantically equivalent to the natural language specification. An unambiguous way to specify the problem is to formulate it as a supervised learning task– given an set of input examples I and output examples O, construct a program p such that for every input $i \in I$, $p(i)$ is in O. This is called programming-by-examples (PBE).

5.1 Programming-by-Examples

For the synthesis of SQL queries, PBE techniques require the user to provide small input and output tables that capture the target concept. For example, to learn SQL query Q_0, a user can provide a small set of rows (at least one) from the Papers, Conferences, and Cities tables. In contrast to the techniques discussed in Section 3 and Section 4, PBE techniques offer a way to validate the synthesized query by checking it against the input-output examples and ensure soundness.

However, there are some limitations of this approach. Firstly, the user has to be familiar with the database schema. Secondly, curating the input-output example can be cumbersome when many tables are involved (or when the schema is large). And finally, highly expressive queries require more rows in order to capture user intent. A number of PBE techniques are also limited by their expressiveness as they target only fragments of relational queries. Figure 4 presents a summary of some PBE tools for synthesis of SQL queries.

5.2 Best of Both Worlds

Baik et. al. [1] propose dual-specification query synthesis, which combines natural language based synthesis with an (optional) PBE task and implement it as the Duoquest system. This system allows the user to provide the natural language description accompanied by a set of input-output examples and guarantees that the program synthesized from the natural language description will be consistent with the given examples.

Consider the task of finding counties of publication for each paper in Papers table from Section 1. With Duoqest, the user can provide a PBE-like specification called a table sketch query (TSQ). Figure 5 shows the TSQ for this case. It has four components: (1) type annotations, (2) a list of example output tuples (this is a subset of the desired output table

Types	text	text
Tuples		
1.	Coarse-to-Fine Decoding for Neural Semantic Parsing	Australia
2.	Duoquest: A Dual-Specification System for Expressive SQL Queries	USA
Sorted?	No	
Limit?	None	

Fig. 5. Example table sketch query (TSQ) for the running example. The top row contains data types for each column, middle rows contain example tuples in the output table, and bottom rows indicate if the desired query output will be sorted or limited to top-k tuples.

and need not be exhaustive), (3) a boolean indicating if or not the query has sorted results, and (4) and integer $k \geq 0$ to indicate whether the query should be limited to top-k rows.

Then, for an input database D, a given query q is said to be consistent with a given a TSQ T if the following four conditions are met:

(1) if there exists a projection π of the result of q on D such that the types match the annotation in T,
(2) for each example tuple in T, there exists a distinct tuple in the result of q on D that matches it,
(3) if T requires sorted output, then q must include a sorting operator and produce tuples in (2) in the same order as examples in T, and
(4) if $k > 0$, then q must return at most k tuples.

5.3 The Algorithm

Duoquest also uses partial queries as an intermediate enumeration step. Algorithm 1 describes the Guided Partial Query Enumeration process, which takes in the natural language description N, the table sketch query T, and the database D. It uses a priority queue P initialized to a singleton set with an empty query. The algorithm proceeds iteratively by considering the query p with the highest priority in P and uses a guided enumeration technique to generate a set of candidate queries Q using it. We keep only the queries $q \in Q$ that are consistent with the table sketch query T. At any point if we find a query q that is both, complete and consistent with the table sketch query T, we return it.

Algorithm 1 The Guided Partial Query Enumeration (GPQE) for dual-specification query synthesis. Given a natural language description N, an enumeration guidance model M, the table sketch query T, and the database D, GPQE returns a query q corresponding to N that is consistent with T.

```
(1) Let $P = \{(\emptyset, 1)\}$ be a priority list.
(2) While $P$ is non-empty,
  (a) Let $p$ be the highest priority element in $P$.
  (b) Let $Q = \texttt{EnumNextStep}(p, N, M, D)$ be the list of candidate queries generated from $p$.
  (c) For $q \in Q$:
    (i) If $q$ satisfies the $T$ as described in Section 5.2,
      (A) If $q$ is complete then return $q$ as a candidate query.
      (B) Otherwise, push $(q, \texttt{pr}(q))$ onto $P$.
```

The `EnumNextStep` operation is carried out using a modified version of SyntaxSQLNetsystem [25]. SyntaxSQL-NetUses a collection of recursive neural network modules for making an enumeration decision for a specific SQL syntax element, that is, for instance, if a `WHERE` clause is being predicted, the neural network predicts the column COL, the operator OP, and a term to compare TERM. Each of these predictions allow the query to branch and give new (partial or complete queries). In its default setting, SyntaxSQLNet produces only the query with the highest probability, however, for Duoquest, it was modified to return a set of possible outputs.

SyntaxSQLNet produces a rank for each candidate query with respect to its siblings in the search space by using the softmax function to produce a confidence score in $(0, 1)$ for each output class. This confidence score is used to assign a confidence score to $\texttt{pr}(q)$ to q ensuring the property that the sum of the confidence scores of all child queries of a query q is equal to the confidence score of q.

The final step in this algorithm is to check if a candidate query q satisfies the table sketch query T. The authors propose a technique to prune partial queries that cannot lead to a complete query that satisfies T using a number of heuristics. The method proceeds by checking if the SQL queries contains redundancies as catalogued in [4]. Then, it also checks if the query does not require sorting but uses an `ORDER BY` clause, if the types in T match the column types in the `SELECT` clauses, and if the output values of partial queries are present as tuples in T. These checks accelerate the search while ensuring soundness.

5.4 Evaluation

The tool is evaluated to understand if the dual-specification approach helps end-users to correctly synthesize SQL query better compared to single-specification approaches. For this, Duoquest is compared with SyntaxSQLNet and SQuID. SQuID is a PBE system that makes an open-world assumption and does not require schema knowledge of the user.

The evaluation was run with an interaction model. The user uses a synthesizer to generate a candidate query and if it does not match their intention, they refine the specification. In comparison to SyntaxSQLNet which could complete only 15/64 tasks, Duoquest was successful on 55/64 tasks. Additionally, the mean number of examples provided to Duoquest were less than 1.5, implying that dual-specification outperforms only natural language specification even in cases where there are few output tuples.

In comparison to SQuID, Duoquest performed worse on simple tasks and marginally better on the harder tasks (such as those which contain aggregate operations). The authors suggest that Duoquest may be preferred in cases when users know fewer examples and if they can articulate a natural language description instead.

The paper also discusses a simulation study to understand how each component of the algorithm contributes to its performance, and how the amount of detail provided in the table sketch query affects the performance. In summary, the performance of the algorithm suffers immensely when the guided enumeration or pruning of partial queries is disabled. The performance changes only slightly when the provided output table is partial or incomplete. This suggests that even a small number of tuples can significantly improve performance. Additionally providing type annotations for each column allows shows a significant improvement compared to SyntaxSQLNet that does not use any user-provided type annotations.

5.5 Discussion

This paper rests on the argument that combining the two paradigms of program synthesis allows us to tackle the limitations of both. The evaluation section seems to suggest only one aspect of it− dual-specification outperforms natural language specifications but does not necessarily outperform PBE based techniques such as SQuID. While, most other PBE techniques do not support the open-world assumption, SQuIDhas limitations of its own such as the need for an exhaustive output table. In that regard, Duoquest does offer some advantage.

Secondly, the scope of the synthesis technique is limited to select-project-join-aggregate (SPJA) queries (including grouping, sorting, and limit operators). However, it does not support nested expressions with different logical operators. The joins considered in the implementation are limited to inner joins on foreign key-primary key relationships.

Thirdly, the evaluation section studies various aspects of the dual-specification technique and is wide in scope. However, the interactive model used in the evaluation assumes that the user is proficient enough to interpret the output SQL query and judge if it meets their intent. A strong case for natural language specifications relies on the lack of user's proficiency with SQL, making this evaluation study an experiment far from application.

Finally, the high-level idea in the paper is a modular approach to synthesis and one can adapt these ideas to related domains of program synthesis, as well as develop the possibility of multi-specification synthesis techniques beyond the combination described here. We discuss this further in Section 6.

6 CONCLUSION

As discussed in Section 2, synthesis of programs from natural language descriptions have been an active area of research. In this report I summarize three methods for synthesis of SQL queries from natural language descriptions that are motivated by different techniques such as neural refinement, type-driven refinement and program repair, and programming-by-examples. Surveying these papers reveal three goals of a good technique:

(1) A learned program should be expressive. The target language should be as less constrained as possible.
(2) The tool should not seek minimal guidance from the user beyond the natural language description. In particular, the tool should not assume that the user is aware of the database schema.
(3) The learned programs should be accurate and match user intent.

In terms of expressive power, the three target SQL with minor variations. The neural refinement technique focuses on WikiSQL, which is a useful but restricted fragment of SQL and does not display features such as nested operators. Therefore, it is the most restrictive of the three, followed by SQLizer which supports most common operators (as defined in the extended relational algebra) but not the full power of SQL. With the least restrictive language, Duoquest is the most expressive tool.

However, the scope of Duoquest goes against the idea of seeking minimal guidance from the user. It not only seeks additional input-output examples, but also details such as if the output should be sorted or if any limits are used in the query. Additionally, the tool is evaluated in an interactive setting, implying that the user plays a greater role in this paradigm and it is farthest away from push-button synthesis. SQLizer does not use user guidance, however, needs domain-specific heuristics, making the neural refinement technique the one needing least user guidance.

One can observe the obvious trend here that expressive power is negatively correlated with user guidance, begging two questions:

(1) Whether there is always a compromise between expressive power and the need for user guidance, and
(2) Where is the sweet-spot that balances expressive power and user-guidance.

Recent work in programming-by-examples such as Example-Guided Synthesis [21] and GenSynth [13] open up a possibility of having highly expressive tools that use minimal user guidance, and invites an exploration of similar techniques in synthesis from natural language descriptions.

The sweet-spot that balances expressive power and user-guidance is very specific to the application of the synthesis tool. However, as discussed, the evaluation of Duoquest in the interactive setting suggests that a user of the tool must be proficient in SQL to determine if the output of the program is as intended.

Duoquest also suggests an interactive line of work to use a combination of specifications to enhance synthesis. This leads to two lines of future work:

(1) A programming-by-example technique with support for optional natural language description that can be used to refine the search,
(2) Using dual specification in domains beyond program synthesis, such as for reactive synthesis.

I would like to believe that learning methods and techniques from different domains will help advance program synthesis and allow us to tackle the holy grail of artificial intelligence.

REFERENCES

[1] Christopher Baik, Zhongjun Jin, Michael Cafarella, and H. V. Jagadish. 2020. Duoquest: A Dual-Specification System for Expressive SQL Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2319–2329. https://doi.org/10.1145/3318464.3389776

[2] Fuat Basik, Benjamin Hättasch, Amir Ilkhechi, Arif Usta, Shekar Ramaswamy, Prasetya Utama, Nathaniel Weir, Carsten Binnig, and Ugur Cetintemel. 2018. DBPal: A Learned NL-Interface for Databases. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1765–1768. https://doi.org/10.1145/3183713.3193562

[3] 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*. Association for Computational Linguistics, Seattle, Washington, USA, 1533–1544. https://aclanthology.org/D13-1160

[4] Stefan Brass and Christian Goldberg. 2006. Semantic errors in SQL queries: A quite complete list. *Journal of Systems and Software* 79, 5 (May 2006), 630–644. https://doi.org/10.1016/j.jss.2005.06.028

[5] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (June 1970), 377–387. https://doi.org/10.1145/362384.362685

[6] Li Dong and Mirella Lapata. 2018. Coarse-to-Fine Decoding for Neural Semantic Parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Melbourne, Australia, 731–742. https://doi.org/10.18653/v1/P18-1068

[7] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (nov 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[8] Srini Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a Neural Semantic Parser from User Feedback. *ArXiv* abs/1704.08760 (2017).

[9] Fei Li and H. V. Jagadish. 2014. Constructing an Interactive Natural Language Interface for Relational Databases. *Proc. VLDB Endow.* 8, 1 (sep 2014), 73–84. https://doi.org/10.14778/2735461.2735468

[10] Fei Li and H. V. Jagadish. 2014. Constructing an Interactive Natural Language Interface for Relational Databases. *Proc. VLDB Endow.* 8, 1 (Sept. 2014), 73–84. https://doi.org/10.14778/2735461.2735468

[11] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. arXiv:arXiv:1508.04025

[12] 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*. Association for Computational Linguistics, Baltimore, Maryland, 55–60. https://doi.org/10.3115/v1/P14-5010

[13] Jonathan Mendelson, Aaditya Naik, Mukund Raghothaman, and Mayur Naik. 2021. GENSYNTH: Synthesizing Datalog Programs without Language Bias. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 7 (May 2021), 6444–6453. https://ojs.aaai.org/index.php/AAAI/article/view/16799

[14] Ana-Maria Popescu, Alex Armanasu, Oren Etzioni, David Ko, and Alexander Yates. 2004. Modern Natural Language Interfaces to Databases: Composing Statistical Parsing with Semantic Tractability. In *Proceedings of the 20th International Conference on Computational Linguistics* (Geneva, Switzerland) *(COLING '04)*. Association for Computational Linguistics, USA, 141–es. https://doi.org/10.3115/1220355.1220376

[15] 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* (Miami, Florida, USA) *(IUI '03)*. Association for Computing Machinery, New York, NY, USA, 149–157. https://doi.org/10.1145/604045.604070

[16] 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. *Proc. VLDB Endow.* 9, 12 (aug 2016), 1209–1220. https://doi.org/10.14778/2994509.2994536

[17] 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. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 1209–1220. https://doi.org/10.14778/2994509.2994536

[18] Armando Solar Lezama. 2008. *Program Synthesis By Sketching*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html

[19] Yibo Sun, Duyu Tang, Nan Duan, Jianshu Ji, Guihong Cao, Xiaocheng Feng, Bing Qin, Ting Liu, and Ming Zhou. 2018. Semantic Parsing with Syntax- and Table-Aware SQL Generation. arXiv:arXiv:1804.08338

[20] Lappoon R. Tang and Raymond J. Mooney. 2000. Automated Construction of Database Interfaces: Intergrating Statistical and Relational Learning for Semantic Parsing. In *2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*. Association for Computational Linguistics, Hong Kong, China, 133–141. https://doi.org/10.3115/1117794.1117811

[21] Aalok Thakkar, Aaditya Naik, Nathaniel Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. 2021. *Example-Guided Synthesis of Relational Queries*. Association for Computing Machinery, New York, NY, USA, 1110–1125. https://doi.org/10.1145/3453483.3454098

[22] William A. Woods. 1977. Lunar Rocks in Natural English: Explorations in Natural Language Question Answering. In *Linguistic Structures Processing*, Antonio Zampolli (Ed.). North Holland, Amsterdam, 521–569.

[23] Xiaojun Xu, Chang Liu, and Dawn Song. 2017. SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning. arXiv:arXiv:1711.04436

[24] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query Synthesis from Natural Language. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 63 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133887

[25] Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. 2018. SyntaxSQLNet: Syntax Tree Networks for Complex and Cross-Domain Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Brussels, Belgium, 1653–1663. https://doi.org/10.18653/v1/D18-1193

[26] John M. Zelle and Raymond J. Mooney. 1996. Learning to Parse Database Queries Using Inductive Logic Programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2* (Portland, Oregon) *(AAAI'96)*. AAAI Press, 1050–1055.

[27] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. arXiv:arXiv:1709.00103