SQLizer: Query Synthesis from Natural Language

NAVID YAGHMAZADEH, The University of Texas at Austin, USA
YUEPENG WANG, The University of Texas at Austin, USA
ISIL DILLIG, The University of Texas at Austin, USA
THOMAS DILLIG, The University of Texas at Austin, USA

This paper presents a new technique for automatically synthesizing SQL queries from natural language (NL). At the core of our technique is a new NL-based program synthesis methodology that combines semantic parsing techniques from the NLP community with type-directed program synthesis and automated program repair. Starting with a program sketch obtained using standard parsing techniques, our approach involves an iterative refinement loop that alternates between quantitative type inhabitation and automated sketch repair. We use the proposed idea to build an end-to-end system called SQLIZER that can synthesize SQL queries from natural language. Our method is fully automated, works for any database without requiring additional customization, and does not require users to know the underlying database schema. We evaluate our approach on over 450 natural language queries concerning three different databases, namely MAS, IMDB, and YELP. Our experiments show that the desired query is ranked within the top 5 candidates in close to 90% of the cases and that SQLIZER outperforms NALIR, a state-of-the-art tool that won a best paper award at VLDB'14.

CCS Concepts: • **Human-centered computing** → **Natural language interfaces**; • **Software and its engineering** → **Domain specific languages**; • **Information systems** → *Relational database query languages*; *Database utilities and tools*; *Extraction, transformation and loading*;

Additional Key Words and Phrases: Program Synthesis, Programming by Natural Languages, Relational Databases.

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

1 INTRODUCTION

In recent years, there has been considerable interest in automated synthesis of programs from informal specifications, such as input-output examples [Barowy et al. 2015; Feser et al. 2015; Gulwani 2011; Le and Gulwani 2014; Osera and Zdancewic 2015; Polozov and Gulwani 2015; Yaghmazadeh et al. 2016]. Such computer-aided programming techniques have been successful in a wide range of domains, ranging from spreadsheets [Barowy et al. 2015; Gulwani 2011; Le and Gulwani 2014] to XML documents [Le and Gulwani 2014; Yaghmazadeh et al. 2016] to R programming [Feng et al. 2017a].

A particularly promising application domain for such computer-aided programming techniques is the automated synthesis of database queries. Although many end-users need to query data stored

Authors' email addresses are (nyaghma, ypwang, isil, tdillig)@cs.utexas.edu.

2475-1421/2017/10-ART63
https://doi.org/10.1145/3133887

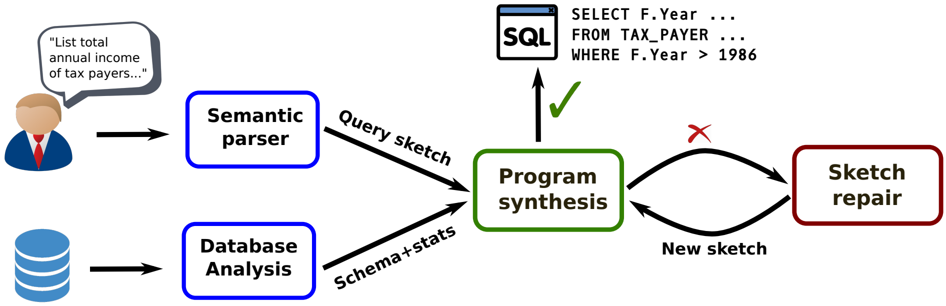

Fig. 1. Schematic overview of our approach

in some relational database, they typically lack the expertise to write complex queries in declarative query languages such as SQL. As a result, there has been a flurry of interest in automatically synthesizing SQL queries from informal specifications [Feng et al. 2017a; Li and Jagadish 2014; Popescu et al. 2003; Wang et al. 2017; Zhang and Sun 2013].

Existing techniques for automatically synthesizing SQL queries fall into two different classes, namely example-based approaches and those based on natural language. Programming-by-example techniques, such as Scythe [Wang et al. 2017] and SQLSynthesizer [Zhang and Sun 2013], require the user to present a miniature version of the database together with the expected output. A shortcoming of such example-directed techniques is that they require the user to be familiar with the database schema. Furthermore, because realistic databases typically involve many tables, it can be quite cumbersome for the user to express their intent using input-output examples.

On the other end of the spectrum, techniques that can generate SQL queries from natural language (NL) descriptions are easier for end-users but more difficult for the underlying synthesizer, as natural language is inherently ambiguous. Existing NL-based techniques try to achieve high precision either by training the system on a specific database [Tang and Mooney 2000; Zelle and Mooney 1996] or requiring interactive guidance from the user [Li and Jagadish 2014]. Hence, existing NL-based techniques for synthesizing SQL queries are either not fully automatic or not database-agnostic (i.e., require customization for each database).

In this paper, we present a novel technique, and its implementation in a tool called Sqlizer, for synthesizing SQL queries from English descriptions. By marrying ideas from the NLP community with type-directed synthesis and repair techniques from the programming languages community, our proposed approach overcomes many of the disadvantages of existing techniques. Specifically, our method is fully automatic (i.e., it does not require guidance from the user) and database-agnostic (i.e., it does not require database-specific training or customization). As we show experimentally, Sqlizer achieves high precision across multiple different databases and significantly outperforms Nalir [Li and Jagadish 2014], a state-of-the-art system that won a best paper award at VLDB'14.

From a technical perspective, our approach achieves these results by combining three novel and synergistic ideas in a confidence-driven refinement loop:

Sketch generation semantic parsing. As a first step, our method uses standard semantic parsing techniques [Kate and Mooney 2006; Kate et al. 2005; Liang and Potts 2015] from the NLP community to translate the user's English description into a so-called *query sketch (skeleton)*. Since a query sketch only specifies the shape – rather than the full content – of the query (e.g., join followed by selection followed by projection), the semantic parser does not need to be trained on the target database. Hence, by using the semantic parser to generate a query sketch rather than a full-fledged

SQL query, we can translate the user's English description into a suitable formal representation *without requiring any database-specific training*. This property of being database-agnostic is very useful because our system does not need additional training data for each new database that a user wishes to query.

Type-directed sketch completion. Given a query sketch containing holes, our technique employs type-directed program synthesis to complete the sketch into a well-typed SQL query. However, because there are typically many well-typed completions of the sketch, our approach assigns a *confidence score* to each possible completion using both the schema and the contents of the database.

Sketch refinement using repair. Since users are typically not familiar with the underlying data organization, the initial query sketches generated using semantic parsing may not accurately reflect the structure of the target query. Hence, there may not be any well-typed, high-confidence completions of the sketch. For example, consider a scenario in which the user believes that the desired data is stored in a single database table, although it in fact requires joining two different tables. Since the user's English description does not adequately capture the structure of the desired query, the initial query sketch needs to be repaired. Our approach deals with this challenge by (a) performing fault localization to pinpoint the likely cause of the error, and (b) using a database of "repair tactics" to refine the initial sketch.

Figure 1 gives a schematic overview illustrating the interaction between the three key ideas outlined above. Given the user's natural language description, Sqlizer first generates the top k most likely query sketches Q using semantic parsing, and, for each query skeleton $q \in Q$, it tries to synthesize a well-typed, high-confidence completion of q. If no such completions can be found, Sqlizer tries to identify the root cause of failure and automatically repairs the suspect parts of the sketch. Once Sqlizer enumerates all high-confidence queries that can be obtained using at most n repairs on q, it then moves on to the next most-likely query sketch. At the end of this *parse-synthesize-repair* loop, Sqlizer ranks all queries based on their confidence scores and presents the top m results to the user.

The general idea. While our target application domain in this paper is relational databases, we believe that our proposed ideas could be applicable in other domains that require synthesizing programs from natural language descriptions. Specifically, given a program sketch generated using standard NLP techniques, we propose a confidence-driven synthesis methodology that uses a form of *quantitative type inhabitation* to assign a confidence score to each well-typed sketch completion. While the specific technique used for assigning confidence scores is inherently domain-specific, the idea of assigning confidence scores to type inhabitants is not. Given a domain-specific method for performing quantitative type inhabitation and a database of domain-specific repair techniques, the *parse-synthesize-repair* loop proposed in this paper can be instantiated in many other settings where the goal is to synthesize programs from natural language.

Results. We have evaluated our approach on 455 queries involving three different databases, namely MAS (Microsoft Academic Search), IMDB, and YELP. Our evaluation shows that the desired query is ranked within Sqlizer's top 5 solutions approximately 90% of the time and within top 1 close to 80% of the time. We also compare Sqlizer against a state-of-the-art NLIDB tool, Nalir, and show that Sqlizer performs significantly better across all three databases, including on the data set that is used for evaluating Nalir.

Contributions. In summary, this paper makes the following key contributions:

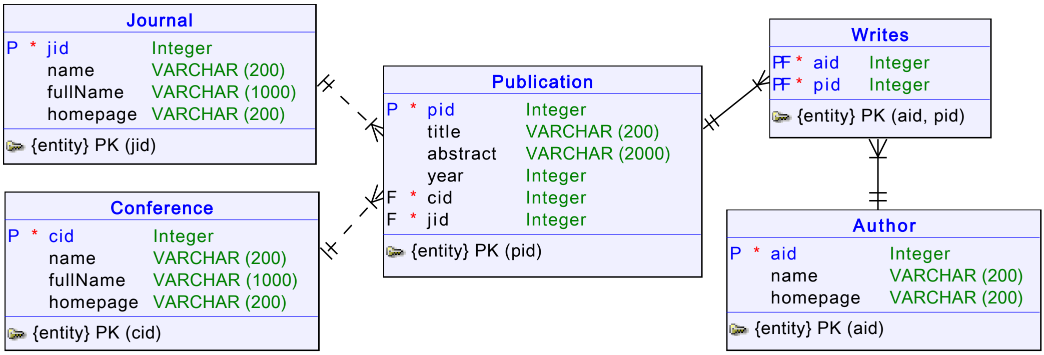

Fig. 2. Simplified schema for the Microsoft Academic Search (MAS) database

- We describe a new methodology for synthesizing programs from natural language. Our synthesis algorithm starts with a program sketch obtained using parsing techniques from the NLP community, and then enters an iterative refinement loop that alternates between type-directed synthesis and automated repair.
- We present Sqlizer, an end-to-end system for synthesizing SQL queries from natural language. Sqlizer is fully automated, database-agnostic, and does not require users to know the underlying database schema.
- We evaluate Sqlizer on a set of 455 queries involving three databases. Our results show that the desired query is ranked number one in 78.4% of benchmarks and within the top 5 in 88.3% of the benchmarks.

Organization. The rest of the paper is organized as follows: We first provide an overview of our approach through a simple motivating example (Section 2). Then, we present our general methodology for synthesizing programs from natural language descriptions (Section 3). After providing some background on a variant of relational algebra (Section 4), we then show how to instantiate each of the components of our general synthesis methodology in the context of SQL synthesis (Sections 5, 6, 7). Finally, we discuss our implementation in Section 8, present our empirical evaluation in Section 9 and survey related work in Section 10.

2 OVERVIEW

In this section, we give a high-level overview of our technique with the aid of a simple motivating example. Figure 2 shows the relevant portion of the schema for the Microsoft Academic Search (MAS) database, and suppose that we would like to synthesize a database query to retrieve the number of papers in OOPSLA 2010. To use our tool, the user provides an English description, such as *"Find the number of papers in OOPSLA 2010"*. We now outline the steps taken by Sqlizer in synthesizing the desired SQL query.

Sketch generation. Our approach first uses a semantic parser to generate the top k most-likely program sketches. For this example, the highest-ranked query sketch returned by the semantic parser is the following [1]:

```
SELECT count(?[papers]) FROM ??[papers] WHERE ? = "OOPSLA 2010"
```

[1] We actually represent query sketches using an extended version of relational algebra. In this section, we present query sketches using SQL for easier readability.

Here, `??` represents an unknown table, and `?`'s represent unknown columns. Where present, the words written in square brackets represent so-called "hints" for the corresponding hole. For example, the second hint in this sketch indicates that the table represented by `??` is semantically similar to the English word "papers".

First iteration. Starting from the above sketch $\mathcal{S}$, Sqlizer enumerates all well-typed completions q_i of $\mathcal{S}$, together with a score for each q_i. In this case, there are many possible well-typed completions of $\mathcal{S}$, however, none of the q_i's meet our confidence threshold. For instance, one of the reasons why Sqlizer fails to find a high-confidence query is that there is no entry called "OOPSLA 2010" in any of the database tables.

Next, Sqlizer performs fault localization on $\mathcal{S}$ to identify the root cause of failure (i.e., not meeting confidence threshold). In this case, we determine that the likely root cause is the predicate `? = "OOPSLA 2010"` since there is no database entry matching "OOPSLA 2010", and our synthesis algorithm has assigned a low confidence score to this term. Next, we repair the sketch by splitting the where clause into two separate conjuncts. As a result, we obtain the following refinement $\mathcal{S}'$ of the initial sketch $\mathcal{S}$:

```
SELECT count(?[papers]) FROM ??[papers]
WHERE ? = "OOPSLA" AND ? = 2010
```

Second iteration. Next, Sqlizer tries to complete the refined sketch $\mathcal{S}'$ but it again fails to find a high-confidence completion of $\mathcal{S}'$. In this case, the problem is that there is no single database table that contains both the entry "OOPSLA" as well as the entry "2010". Going back to fault localization, we now determine that the most likely problem is the term `??[papers]`, and we try to repair it by introducing a join. As a result, the new sketch $\mathcal{S}''$ now becomes:

```
SELECT count(?[papers]) FROM  ??[papers] JOIN  ?? ON ? = ?
WHERE ? = "OOPSLA" AND ? = 2010
```

Third iteration. After going back to the sketch completion phase a third time, we are now able to find a high-confidence instantiation q of $\mathcal{S}''$. In this case, the highest ranked completion of $\mathcal{S}''$ corresponds to the following query:

```
SELECT count(Publication.pid)
FROM Publication JOIN Conference ON Publication.cid = Conference.cid
WHERE Conference.name = "OOPSLA" AND Publication.year = 2010
```

This query is indeed the correct one, and running it on the MAS database yields the number of papers in OOPSLA 2010.

3 GENERAL SYNTHESIS METHODOLOGY

Before describing our specific technique for synthesizing SQL code from English queries, we first explain our general methodology for synthesizing programs from natural language descriptions. As mentioned in Section 1, our general synthesis methodology involves three components, namely (a) sketch generation using semantic parsing, (b) sketch completion using quantitative type inhabitation, and (c) sketch refinement using repair.

The high-level structure of our synthesis methodology is described in pseudo-code in Algorithm 1. The algorithm takes as input a natural language description Q of the program to be synthesized, a type environment Γ, and a confidence threshold γ. Since our synthesis algorithm assigns confidence scores to generated programs, the cut-off γ is used to identify programs that do not meet some minimum confidence threshold. The output of the synthesis procedure is a list of synthesized programs, together with their corresponding confidence score.

Algorithm 1 General synthesis methodology

```
procedure SYNTHESIZE($Q$, $\Gamma$, $\gamma$)
    Input: natural language query $Q$, type environment $\Gamma$, confidence threshold $\gamma$
    Output: A list of synthesized programs and their corresponding confidence scores
    $Sketches$ := SEMANTICPARSE($Q$)                                    ▷ Sketch generation
    $Programs$ := [ ];
    for all top $k$ ranked $\mathcal{S} \in Sketches$ do
        loop $n$ times
            $\theta$ := FINDINHABITANTS ($\mathcal{S}$, $\Gamma$)                ▷ Type-directed sketch completion
            $needRepair$ := $true$
            for all $(\mathcal{I}_i, \mathcal{P}_i) \in \theta$ do
                if $\mathcal{P}_i > \gamma$ then $Programs$.add($\mathcal{I}_i, \mathcal{P}_i$); $needRepair$ := $false$
            if $\neg needRepair$ then break
            $\mathcal{F}$ := FAULTLOCALIZE($\mathcal{S}$, $\Gamma$, $\emptyset$)                   ▷ Sketch refinement
            if $\mathcal{F}$ = null then break
            $\mathcal{S}$ := $\mathcal{S}$[FINDREPAIR($\mathcal{F}$)/$\mathcal{F}$]
    return $Programs$
```

Given an English description of the program, the first step in our synthesis methodology is to generate a ranked list of program sketches using semantic parsing [Kate and Mooney 2006; Kate et al. 2005; Liang and Potts 2015]. Semantic parsing is a standard technique from the NLP community that can be used to convert an English utterance into a logical form, which is defined according to a formal context-free grammar. In this paper, we use *program sketches* [Solar-Lezama et al. 2005, 2006] as our logical form representation because they allow us to translate the NL description into a program draft while omitting several low-level details that may not be accurately captured by the user's English description. As standard, a *program sketch* in this context is a partial program with holes [Solar-Lezama et al. 2005, 2006]. However, because our program sketches are created from English descriptions, we additionally allow each hole in the sketch to contain a natural language *hint* associated with it. The idea is to use these natural language hints for assigning confidence scores during sketch completion.

Given the top k query sketches generated by the semantic parser, the next step is to complete each sketch $\mathcal{S}$ such that it is well-typed with respect to our type environment Γ. For instance, in the context of SQL, the type environment corresponds to the database schema, and the goal of sketch completion is to find a relational algebra term that is well-typed with respect to the underlying database schema. In essence, the use of type information allows our synthesis methodology to perform logical reasoning that complements the probabilistic reasoning performed by the semantic parser. However, because there are typically many well-typed completions of a given sketch, we would like to predict which term is the *most likely* completion. We refer to this problem as *quantitative type inhabitation*: Given a type τ, a type environment Γ, and some "soft constraints" C on the term to be synthesized, what is our confidence $\mathcal{P}_i$ that $\mathcal{I}_i$ is the inhabitant of τ with respect to hard constraints Γ and soft constraints C? These soft constraints include natural language hints embedded inside the sketch as well as any other domain-specific knowledge. For example, in the context of database queries, we also utilize the contents of the database when assigning confidence scores to relational algebra terms.

$$
\begin{array}{rcl}
T & := & \Pi_L(T) \mid \sigma_\phi(T) \mid T\ {}_c\!\bowtie_c T \mid t \\
L & := & L, L \mid c \mid f(c) \mid g(f(c), c) \\
E & := & T \mid c \mid v \\
\phi & := & \phi\ lop\ \phi \mid \neg\phi \mid c\ op\ E \\
op & := & \leq \mid < \mid = \mid > \mid \geq \\
lop & := & \wedge \mid \vee
\end{array}
$$

Fig. 3. Grammar of Extended Relational Algebra. Here, t, c denote table and column names; f denotes an aggregate function, and v denotes a value.

id	name	score	cid_fk
1	John	60	101
2	Jack	80	102
3	Jane	80	103
4	Mike	90	104
5	Peter	100	103
6	Alice	100	104

(a) Grades

cid	cname	dept
101	C1	CS
102	C2	EE
103	C3	CS
104	C4	EE

(b) Courses

Fig. 4. Example tables

Now, if we fail to find any well-typed completions of the sketch that meet our confidence threshold γ, our algorithm goes into a refinement loop that alternates between repair and synthesis (the inner loop at lines 7–15 in Algorithm 1). Given a program sketch $\mathcal{S}$ for which we cannot find a high-confidence completion, the fault localization procedure (line 13) returns a *minimal subterm* of the sketch that does not meet our confidence threshold γ. Given a faulty subterm $\mathcal{F}$ of sketch $\mathcal{S}$, we then consult a database of *domain-specific repair tactics* to find a new subterm $\mathcal{F}'$ that can be used to replace $\mathcal{F}$. For instance, in the domain of SQL query synthesis, the repair tactics introduce join operators, aggregate functions, or additional conjuncts in predicates depending on the shape of the faulty sub-term. If there are multiple repair tactics that apply, we can either arbitrarily choose one or try each of them in turn. Since this refinement process may, in general, continue ad infinitum, our algorithm allows the user to specify a value n that controls the number of refinement steps that are allowed.

4 EXTENDED RELATIONAL ALGEBRA

In the rest of this paper, we will show how to apply the synthesis methodology outlined in Section 3 to the problem of synthesizing database queries from natural language. However, since our approach synthesizes database queries in a variant of relational algebra, we first describe our target language. Note that it is straightforward to translate our extended relational algebra to declarative query languages, such as SQL.

Our target language for expressing database queries is presented in Figure 3. Here, *relations* are denoted as T and include *tables* t stored in the database or *views* obtained by applying the relational algebra operators, projection (Π), selection (σ), and join ($\bowtie$). As standard, projection $\Pi_L(T)$ takes a relation T and a column list L and returns a new relation that only contains the columns in L. The selection operation $\sigma_\phi(T)$ yields a new relation that only contains rows satisfying ϕ in T. The join operation $T_1\ {}_{c_1}\!\bowtie_{c_2} T_2$ composes two relations T_1, T_2 such that the result contains exactly those rows of $T_1 \times T_2$ satisfying $c_1 = c_2$, where c_1, c_2 are columns in T_1, T_2 respectively. Please observe that the grammar from Figure 3 allows nested queries. For instance, selections can occur within other selections and joins as well as inside predicates ϕ.

In the rest of this paper, we make a few assumptions that simplify our technical presentation. First, we assume that every column in the database has a unique name. Note that we can easily enforce this restriction by appending the table name to each column name. Second, we only consider *equi-joins* because they are the most commonly used join operator; however, our techniques can also be extended to other kinds of join operators (e.g., θ-join).

Unlike standard relational algebra, the relational algebra variant shown in Figure 3 also allows aggregate functions as well as a group-by operator. For conciseness, aggregate functions $f \in$

AggrFunc = {max, min, avg, sum, count} are specified as a subscript in the projection operation. In particular, $\Pi_{f(c)}(T)$ yields a single aggregate value obtained by applying f to column c of relation T. Similarly, group-by operations are also specified as a subscript in the projection operator. Specifically, $\Pi_{g(f(c_1),c_2)}(T)$ divides rows of T into groups g_i based on values stored in column c_2 and, for each g_i, it yields the aggregate value $f(c_1)$.

Example 4.1. Consider the "Grades" and "Courses" tables from Figure 4, where column names with suffix "_fk" indicate foreign keys. Here, $\Pi_{avg(score)}(Grades)$ evaluates to 85, and $\Pi_{g(avg(score),\ dept)}(Grades_{cid\_fk} \bowtie_{cid} Courses)$ yields the following table:

dept	avg(score)
CS	80
EE	90

To provide an example of nested queries, suppose that a user wants to retrieve all students with the highest score. We can express this query as $\Pi_{name}(\sigma_{score=\Pi_{max(score)}(Grades)}(Grades))$. For the tables from Figure 4, this query yields a table with two rows, Peter and Alice.

5 SKETCH GENERATION USING SEMANTIC PARSING

Recall from Section 3 that the first step of our synthesis methodology is to generate a program sketch using semantic parsing. In this section, we provide some background on semantic parsing and describe our parser for generating a query sketch from the user's English description.

Background on semantic parsing. The goal of a semantic parser is to map a sentence in natural language to a so-called *logical form* which represents its meaning. A logical form is an unambiguous artificial language specified using a context-free grammar. Previous work on semantic parsing has used various logical form representations, including lambda calculi [Carpenter 1997], database query languages [Zelle and Mooney 1996], and natural logics [MacCartney and Manning 2009].

Similar to traditional parsers used in compilers, a semantic parser is specified by a context-free grammar and contains a set of reduction rules describing how to derive non-terminal symbols from token sequences. Given an English sentence S, the parse is successful if the designated root non-terminal can be derived from S.

Since semantic parsers deal with natural language, they must overcome two challenges that do not arise in parsers for formal languages. First, multiple English phrases (e.g., "forty two") can correspond to a single token (e.g., 42); so a semantic parser must be capable of recognizing phrases and mapping them accurately into tokens. To address this challenge, semantic parsers typically include a linguistic processing module to analyze the sentence and detect such phrases based on part-of-speech tagging and named entity recognition. Second, since natural language is inherently ambiguous, one can often derive multiple logical forms from an English sentence. Modern semantic parsers address this challenge by using statistical methods to predict the most likely derivation for a given utterance. Given a set of training data consisting of pairs of English sentences and their corresponding logical form, the idea is to train a statistical model to predict the likelihood that a given English sentence is mapped to a particular logical form. Hence, the output of a semantic parser is a list of logical forms x_i, each associated with a probability that the English sentence corresponds to x_i.

SQLIZER***'s Semantic Parser.*** The logical forms used in our synthesis methodology take the form of program sketches containing unknown expressions with natural language hints. In essence, the use of program sketches as our logical form representation allows the semantic parser to generate a high-quality intermediate representation even when we have modest amounts of training data

$$
\begin{array}{rcl}
\chi & := & \Pi_\kappa(\chi) \mid \sigma_\psi(\chi) \mid \chi_{?h}\!\bowtie_{?h}\!\chi \mid ??h \\
\kappa & := & \kappa, \kappa \mid ?h \mid f(?h) \mid g(f(?h), ?h) \\
\eta & := & \chi \mid ?h \mid v \\
\psi & := & \psi \; lop \; \psi \mid \neg\psi \mid ?h \; op \; \eta \\
op & := & \leq \;\mid\; < \;\mid\; = \;\mid\; > \;\mid\; \geq \\
lop & := & \wedge \mid \vee
\end{array}
$$

Fig. 5. Sketch Grammar. Here, v denotes a value, h represents a natural language hint, and f is an aggregate function.

$$
\begin{array}{rcl}
\beta & := & Number\,\mathbb{N} \mid Bool\,\mathbb{B} \mid String\,\mathbb{S} \mid \cdots \\
\tau & := & \beta \mid \{(c_1 : \beta_1), \cdots, (c_n : \beta_n)\} \\
\Gamma & :: & Table \to \tau \\
p & :: & \{v : double \mid 0 \leq v \leq 1\} \\
sim & :: & \mathbb{S} \times \mathbb{S} \to p \\
P_{\bowtie} & :: & Column \times Column \to p \\
P_\phi & :: & Column \times Value \to p
\end{array}
$$

Fig. 6. Symbols used in sketch completion

available. In the context of generating database queries from natural language, the use of query sketches allows our technique to work well *without requiring any database-specific training*.

Figure 5 defines *query sketches* that are used as our logical form representation in the database query synthesis domain. At a high level, a query sketch χ is a relational algebra term with missing table and column names. In particular, $?h$ represents an unknown column with *hint* h, which is just a natural language description of the unknown. Similarly, $??h$ represents an unknown table name with corresponding hint h. If there is no hint associated with a hole, we simply write ? for columns and ?? for tables.

To map English sentences to query sketches, we have implemented our own semantic parser on top of the Sempre framework [Berant et al. 2013], which is a toolkit for building semantic parsers. For the linguistic processor, we leverage the pre-trained models of the Stanford CoreNLP [Manning et al. 2014] library for part-of-speech tagging and named entity recognition. Given an utterance u, our parser generates all possible query sketches $\mathcal{S}_i$ and assigns each $\mathcal{S}_i$ a score that indicates the likelihood that $\mathcal{S}_i$ is the intended interpretation of u. This score is calculated based on a set of pre-defined features. More precisely, given an utterance u and weight vector w, the parser maps each query sketch $\mathcal{S}_i$ to a d-dimensional feature vector $\phi(u, \mathcal{S}_i) \in \mathbb{R}^d$ and computes the likelihood score for $\mathcal{S}_i$ as the weighted sum of its features:

$$
score(u, \mathcal{S}_i) \;=\; \vec{w} \,.\, \phi(u, \mathcal{S}_i) \;=\; \sum_{j=1}^{d} w_j \cdot \phi(u, \mathcal{S}_i)_j
$$

Sqlizer uses approximately 40 features that it inherits from the Sempre framework. Examples of features include the number of grammar rules used in the derivation, the length of the matched input, whether a particular rule was used in the derivation, the number of skipped words in a part-of-speech tag etc.

6 TYPE-DIRECTED SKETCH COMPLETION

Given a program sketch $\mathcal{S}$ and a type environment (in our case, database schema) Γ, sketch completion refers to the problem of finding an expression e such that e is well-typed with respect to Γ. In essence, this is a type inhabitation problem in that we can view each program sketch as defining a type. However, rather than finding *any* inhabitant of $\mathcal{S}$, we would like to find an inhabitant e of $\mathcal{S}$ that has a high probability of being the program that the user has in mind. One of the key ideas in this paper is to use natural language hints embedded in sketch $\mathcal{S}$ as well as any domain-specific background knowledge to associate a confidence score $\mathcal{P}_i$ with every inhabitant $\mathcal{I}_i$ of a type τ with respect to the type environment Γ. As mentioned earlier, we refer to this problem as *quantitative type inhabitation*.

In the specific case of the database query synthesis problem, we make use of the following high-level insights to determine the confidence score of each type inhabitant:

- *Names of schema elements:* Since our query sketches contain natural language hints for each hole, we can utilize table and column names in the database schema to assign confidence scores.
- *Foreign and primary keys:* Since foreign keys provide links between data in two different database tables, join operations that involve foreign keys have a higher chance of being the intended term.
- *Database contents:* Our approach also uses the contents of the database when assigning scores to queries. For instance, a candidate term $\sigma_\phi(T)$ is relatively unlikely to occur in the target query if there are no entries in relation T satisfying predicate ϕ.[2]

Using these domain-specific heuristics in the query synthesis context, Figures 7 and 8 describe our quantitative type inhabitation rules. Specifically, the top-level sketch completion rules derive judgments of the form $\Gamma \vdash \chi \Downarrow T : \tau, p$ where Γ is a type environment mapping each table t in the database to its corresponding type. The meaning of this judgment is that sketch χ instantiates to a relational algebra term T of type τ with confidence score $p \in [0, 1]$. The higher the score p, the more likely it is that T is a valid completion of sketch χ.

Figure 8 presents the helper rules used for finding inhabitants of so-called *specifiers*. A specifier ω of a sketch χ is any subterm of χ that does not correspond to a relation. For example, the specifier for $\Pi_\kappa(\chi)$ is κ, and the specifier for $\sigma_\psi(\chi)$ is ψ. The instantiation rules for specifiers are shown in Figure 8 using judgements of the form:

$$\Gamma, \tau \vdash_s \omega \Downarrow Z : \tau', p$$

Here, type τ used on the left-hand side of the judgment denotes the type of the table that ω is supposed to qualify. For instance, if the parent term of ω is $\Pi_\omega(\chi)$, then τ represents the type of relation χ. Hence, the meaning of this judgment is that, under the assumption that ω qualifies a relation of type τ, then ω instantiates to a term Z of type τ' with score p.

Inhabitation rules for relation sketches. Let us first consider the quantitative type inhabitation rules from Figure 7. Given a sketch $??h$ indicating an unknown database table with hint h, we can, in principle, instantiate ?? with any table t in the database (i.e., t is in the domain of Γ). However, as shown in the first rule from Figure 7, our approach uses the hint h to compute the likelihood that t is the intended completion of ??. Specifically, we use the *sim* procedure to compute a similarity score between hint h and table name t using Word2Vec [Mikolov et al. 2013], which uses a two-layer neural net to group similar words together in vector-space.

Next, let us consider the *Proj*, *Sel* rules from Figure 7. Given a sketch of the form $\Pi_\kappa(\chi)$ (resp. $\sigma_\psi(\chi)$), we first recursively instantiate the sub-relation χ to T. Now, observe that the columns used in specifiers κ and ψ can only refer to columns in T; hence, we use the type τ of T when instantiating specifiers κ, ψ. Now, assuming χ instantiates to T with score p_1 and κ (resp. ψ) instantiates to L (resp. ϕ) with score p_2, we need to combine p_1 and p_2 to determine a score for $\Pi_L(T)$ (resp. $\sigma_\phi(T)$). Towards this goal, we define an operator $\otimes$ that is used to compose different scores. While there are many possible ways to compose scores, our implementation defines $\otimes$ as the geometric mean of $p_1, \ldots, p_n$:

$$p_1 \otimes \ldots \otimes p_n = \sqrt[n]{p_1 \times \ldots \times p_n}$$

[2] This assumption may not hold in all contexts. However, since Sqlizer is intended as a question answering system, we believe this assumption makes sense under potential deployment scenarios of a tool like Sqlizer.

$$\frac{\begin{array}{c} t \in dom(\Gamma) \\ p = sim(h, t) \end{array}}{\Gamma \vdash ??h \Downarrow t : \Gamma(t),\ p}\ \text{(Table)} \qquad \frac{\begin{array}{c} \Gamma \vdash \chi \Downarrow T : \tau,\ p_1 \\ \Gamma, \tau \vdash_s \psi \Downarrow \phi : \mathbb{B},\ p_2 \end{array}}{\Gamma \vdash \sigma_\psi(\chi) \Downarrow \sigma_\phi(T) : \tau,\ p_1 \otimes p_2}\ \text{(Sel)}$$

$$\frac{\begin{array}{c} \Gamma \vdash \chi \Downarrow T : \tau,\ p_1 \\ \Gamma, \tau \vdash_s \kappa \Downarrow L : \tau_1,\ p_2 \end{array}}{\Gamma \vdash \Pi_\kappa(\chi) \Downarrow \Pi_L(T) : \tau_1,\ p_1 \otimes p_2}\ \text{(Proj)} \qquad \frac{\begin{array}{c} \Gamma \vdash \chi_1 \Downarrow T_1 : \tau_1,\ p_1 \\ \Gamma \vdash \chi_2 \Downarrow T_2 : \tau_2,\ p_2 \\ \Gamma, \tau_1 \vdash_s ?h_1 \Downarrow c_1 : \{(c_1, \beta)\},\ p_3 \\ \Gamma, \tau_2 \vdash_s ?h_2 \Downarrow c_2 : \{(c_2, \beta)\},\ p_4 \\ p = p_1 \otimes p_2 \otimes p_3 \otimes p_4 \otimes P_{\bowtie}(c_1, c_2) \end{array}}{\Gamma \vdash \chi_1 {}_{?h_1}\!\bowtie_{?h_2} \chi_2 \Downarrow T_1 {}_{c_1}\!\bowtie_{c_2} T_2 : \tau_1 \cup \tau_2,\ p}\ \text{(Join)}$$

Fig. 7. Inference rules for relations

Observe that our definition of ⊗ is not the standard way to combine probabilities (i.e., standard multiplication). We have found that this definition of ⊗ works better in practice, as it does not penalize long queries and allows a more meaningful comparison between different-length queries. However, one implication of this choice is that the scores of all possible completions do not add up to 1. Hence, our confidence scores are not "probabilities" in the technical sense, although they are in the range $[0, 1]$.

Finally, let us consider the *Join* rule for completing sketches of the form $\chi_1 {}_{?h_1}\!\bowtie_{?h_2} \chi_2$. As before, we first complete the nested sketches χ_1 and χ_2, and then instantiate $?h_1$ and $?h_2$ under the assumption that χ_1, χ_2 have types τ_1, τ_2 respectively. The function $P_{\bowtie}(c_1, c_2)$ used in the *Join* rule assigns a higher score to term $T_1 {}_{c_1}\!\bowtie_{c_2} T_2$ if column c_1 is a foreign key referring to column c_2 in table T_2 (or vice versa). Intuitively, if c_1 in table T_1 is a foreign key referring to c_2 in Table T_2, then there is a high probability that the term $T_1 {}_{c_1}\!\bowtie_{c_2} T_2$ appears in the target query. More precisely, we define the $P_{\bowtie}$ function as follows, where ϵ is a small, non-zero constant:

$$P_{\bowtie}(c_1, c_2) = \begin{cases} 1 - \epsilon & \text{if } c_1 \text{ is a foreign key referring to } c_2 \text{ (or vice versa)} \\ \epsilon & \text{otherwise} \end{cases}$$

Inhabitation rules for specifiers. In our discussion so far, we ignored how to find inhabitants of specifiers and how to assign confidence scores to them. We now consider the rules from Figure 8 that address this problem.

In the simplest case, a specifier is a column of the form $?h$. As shown in the *Col* rule of Figure 8, we must first ensure that the candidate inhabitant c is actually a column of the table associated with the parent relation (i.e., $(c, \beta) \in \tau$). In addition to this "hard constraint", we also want to assign a higher confidence score to inhabitants c that have a close resemblance to the natural language hint h provided in the sketch. Hence, similar to the Table rule from Figure 7, we assign a confidence score by computing the similarity between c and h using Word2Vec.

Since most of the rules in Figure 8 are quite similar to the ones from Figure 7, we do not explain them in detail. However, we would like to draw the reader's attention to the *Pred* rule for instantiating predicate sketches of the form $?h\ op\ \eta$. Recall that a predicate $c\ op\ v$ evaluates to true for exactly those values v' in column c for which the predicate $v'\ op\ v$ is true. Now, if c does not contain any entries v' for which the $v'\ op\ v$ evaluates to true, there is a low, albeit non-zero, probability that $c\ op\ v$ is the intended predicate. To capture this intuition, the *Pred* rule uses the following P_ϕ function when assigning a confidence score:

$$\frac{(c,\beta) \in \tau \qquad p = sim(h,c)}{\Gamma, \tau \vdash_s ?h \Downarrow c : \{(c,\beta)\},\ p} \quad \text{(Col)}$$

$$\frac{\Gamma, \tau \vdash_s ?h \Downarrow c : \{(c,\beta)\},\ p_1 \qquad \Gamma, \tau \vdash_s \eta \Downarrow E : \{(c',\beta)\},\ p_2 \qquad p = p_1 \otimes p_2 \otimes P_\phi(c\ op\ E)}{\Gamma, \tau \vdash_s ?h\ op\ \eta \Downarrow c\ op\ E : \mathbb{B},\ p} \quad \text{(Pred)}$$

$$\frac{p = P_\tau(v,\beta),\ \ (c,\beta) \in \tau \qquad cast(v,\beta) = v'}{\Gamma, \tau \vdash_s v \Downarrow v' : \{(v',\beta)\},\ p} \quad \text{(Value)}$$

$$\frac{\Gamma \vdash \chi \Downarrow T : \tau, p}{\Gamma, \tau \vdash_s \chi \Downarrow T : \tau, p} \quad \text{(Reduce)}$$

$$\frac{\Gamma, \tau \vdash_s \psi \Downarrow \phi : \mathbb{B},\ p}{\Gamma, \tau \vdash_s \neg\psi \Downarrow \neg\phi : \mathbb{B},\ p} \quad \text{(PredNeg)}$$

$$\frac{\Gamma, \tau \vdash_s ?h \Downarrow c : \{(c,\beta)\},\ p \qquad type(f) = \beta \rightarrow \beta'}{\Gamma, \tau \vdash_s f(?h) \Downarrow f(c) : \{(f(c),\beta')\},\ p} \quad \text{(Fun)}$$

$$\frac{\Gamma, \tau \vdash_s f(?h_1) \Downarrow f(c_1) : \{(f(c_1),\beta)\},\ p_1 \qquad \Gamma, \tau \vdash_s ?h_2 \Downarrow c_2 : \{(c_2,\tau_2)\},\ p_2}{\Gamma, \tau \vdash_s g(f(?h_1), ?h_2) \Downarrow g(f(c_1), c_2) : \{(c_2,\tau_2), (f(c_1),\beta)\},\ p_1 \otimes p_2} \quad \text{(Group)}$$

$$\frac{\Gamma, \tau \vdash_s \kappa_1 \Downarrow L_1 : \tau_1,\ p_1 \qquad \Gamma, \tau \vdash_s \kappa_2 \Downarrow L_2 : \tau_2,\ p_2}{\Gamma, \tau \vdash_s \kappa_1, \kappa_2 \Downarrow L_1, L_2 : \tau_1 \cup \tau_2,\ p_1 \otimes p_2} \quad \text{(ColList)}$$

$$\frac{\Gamma, \tau \vdash_s \psi_1 \Downarrow \phi_1 : \mathbb{B},\ p_1 \qquad \Gamma, \tau \vdash_s \psi_2 \Downarrow \phi_2 : \mathbb{B},\ p_2}{\Gamma, \tau \vdash_s \psi_1\ lop\ \psi_2 \Downarrow \phi_1\ lop\ \phi_2 : \mathbb{B},\ p_1 \otimes p_2} \quad \text{(PredLop)}$$

Fig. 8. Inference rules for specifiers

$$P_\phi(c\ op\ E) = \begin{cases} 1 - \epsilon & \text{if } \exists v' \in contents(c).\ v'\ op\ E = \top \\ \epsilon & \text{otherwise} \end{cases}$$

Here, ϵ is a small, non-zero constant that indicates low confidence. Hence, predicate $c\ op\ E$ is assigned a low score if there are no entries satisfying it in the database. [3]

Finally, we would also like to draw the reader's attention to the *Value* rule, which types constants in a quantitative manner. To gain intuition about the necessity of quantitatively typing constants, consider the string constant "forty two". While this value may correspond to a string constant, it could also be an integer (42) or a float (42.0). To deal with such ambiguity, our typing rules use a function P_τ to estimate the probability that constant v has type β and then cast v to a constant v' of type β.

Example 6.1. Consider again the (tiny) database shown in Figure 4 and the following query sketch:

$$\Pi_{g(avg(?score),\ ?department)}(??score\ {}_{?}\!\bowtie_{?}\, ??)$$

According to the rules from Figures 7 and 8, the query $\Pi_{g(avg(score),\ dept)}(Grades_{cid\_fk} \bowtie_{cid\_fk} Grades)$ is not a valid completion because it is not "well-typed" (i.e., `dept` is not a column in $Grades_{cid\_fk} \bowtie_{cid\_fk} Grades$). In contrast, the query $\Pi_{g(avg(score),\ dept)}(Grades_{id} \bowtie_{cid} Courses)$ is well-typed but is assigned a low score because the column `id` in Grades is not a foreign key referring to column `cid` in Courses. As a final example, consider the query:

$$\Pi_{g(avg(score),\ dept)}(Grades_{cid\_fk} \bowtie_{cid} Courses)$$

[3] Observe that this heuristic requires querying the underlying database. Hence, if Sqlizer is used as part of an online system that has direct access to the live database, our synthesis algorithm may place a load on the database by issuing multiple queries in short succession. However, this problem can be avoided by forking the database rather than using the live version.

Algorithm 2 Fault Localization Algorithm

```
procedure FaultLocalize($\mathcal{S}$, $\Gamma$, $\tau$)
    Input: partial sketch $\mathcal{S}$, schema $\Gamma$, record type $\tau$
    Output: faulty partial sketch $\mathcal{S}'$ or null
    if isRelation($\mathcal{S}$) then
        for all $(\chi_i, \omega_i) \in$ SubRelations($\mathcal{S}$) do
            $\chi_i'$ = FaultLocalize($\chi_i$, $\Gamma$, $\tau$)
            if $\chi_i' \neq$ null then return $\chi_i'$
            $\theta_i$ = FindInhabitants($\chi_i$, $\Gamma$)
            for all $(\mathcal{I}_j, \mathcal{P}_j, \tau_j) \in \theta_i$ do
                $\omega_{ij}'$ = FaultLocalize($\omega_i$, $\Gamma$, $\tau_j$)
            if $\forall j.\ \omega_{ij}' \neq$ null then
                if $\forall j, k.\ \omega_{ij}' = \omega_{ik}'$ then return $\omega_{i0}'$
                else if CanRepair($\omega_i$) then return $\omega_i$
    else if isSpecifier($\mathcal{S}$) then
        for all $\omega_i$ $\in$SubSpecifiers($\mathcal{S}$) do
            $\omega_i'$ = FaultLocalize($\omega_i$, $\Gamma$, $\tau$)
            if $\omega_i' \neq$ null then return $\omega_i'$
    $\theta$ = FindInhabitants($\mathcal{S}$, $\Gamma$, $\tau$)
    if MaxProb($\theta$) < $\rho$ and CanRepair($\mathcal{S}$) then
        return $\mathcal{S}$
    else return null
```

This query is both well-typed and is assigned a high score close to 1.

7 SKETCH REFINEMENT USING REPAIR

In the previous section, we saw how to generate a ranked list of possible sketch completions using quantitative type inhabitation. However, in some cases, it may not be possible to find well-typed, high-confidence completions of *any* sketch. For instance, this situation can arise for at least two different reasons:

(1) Due to ambiguities in the user's natural language description, the correct sketch may not be in the top k sketches generated by the semantic parser.
(2) In some cases, the user's natural language description may be misleading. For instance, in the context of query synthesis, the user might make incorrect assumptions about the underlying data organization , so her English description may not accurately reflect the general structure of the target query.

One of the key ideas underlying our synthesis methodology is to overcome these problems through the use of *automated sketch refinement.* Given a faulty sketch $\mathcal{S}$, the goal of sketch refinement is to generate a new program sketch $\mathcal{S}'$ such that $\mathcal{S}'$ repairs a potentially faulty sub-part of $\mathcal{S}$. Similar to prior approaches on automated program repair, our method performs sketch refinement using a combination of *fault localization* and a database of *repair tactics.* However, since we do not have access to a concrete test case that exhibits a specific bug, our sketch refinement procedure is again confidence-driven. Specifically, we perform fault localization by identifying a *minimal fault subpart* $\mathcal{F}$ of the sketch such that $\mathcal{F}$ does not have any high-confidence inhabitants. The *minimal*

$$\begin{array}{rcl@{\qquad}rcl}
\textsc{SubRelations}(\Pi_\kappa(\chi)) & = & \{(\chi,\kappa)\} & \textsc{SubSpecifiers}(\kappa_1,\kappa_2) & = & \{\kappa_1,\kappa_2\} \\
\textsc{SubRelations}(\sigma_\psi(\chi)) & = & \{(\chi,\psi)\} & \textsc{SubSpecifiers}(?h\ op\ \eta) & = & \{?h,\eta\} \\
\textsc{SubRelations}(\chi_1\ {}_{?h_1}\!\bowtie_{?h_2}\chi_2) & = & \{(\chi_1,?h_1),(\chi_2,?h_2)\} & \textsc{SubSpecifiers}(\neg\psi) & = & \{\psi\} \\
\textsc{SubSpecifiers}(g(f(?h_1),?h_2)) & = & \{f(?h_1),?h_2\} & \textsc{SubSpecifiers}(\psi_1\ lop\ \psi_2) & = & \{\psi_1,\psi_2\}
\end{array}$$

Fig. 9. Auxiliary functions used in Algorithm 2

$$\dfrac{split(v) = (v_1, v_2),\quad v_2 \neq \epsilon}{?h\ op\ v \rightsquigarrow ?h\ op\ v_1 \wedge ?h\ op\ v_2}\ (\text{AddPred}) \qquad \dfrac{}{\chi_1\ {}_{?h_1}\!\bowtie_{?h_2}\chi_2 \rightsquigarrow \chi_1\ {}_{?\epsilon}\!\bowtie_{?\epsilon} ??\epsilon\ {}_{?\epsilon}\!\bowtie_{?\epsilon}\chi_2}\ (\text{AddJoin3})$$

$$\dfrac{}{\sigma_\psi(\chi) \rightsquigarrow \sigma_\psi(\chi\ {}_{?\epsilon}\!\bowtie_{?\epsilon} ??\epsilon)}\ (\text{AddJoin1}) \qquad \dfrac{\begin{array}{c} split(h) = (f', h') \\ f \in AggrFunc,\quad sim(f, f') \geq \delta \end{array}}{?h \rightsquigarrow f(?h')}\ (\text{AddFunc})$$

$$\dfrac{}{\Pi_\kappa(\chi) \rightsquigarrow \Pi_\kappa(\chi\ {}_{?\epsilon}\!\bowtie_{?\epsilon} ??\epsilon)}\ (\text{AddJoin2}) \qquad \dfrac{}{?h\ op\ v \rightsquigarrow ?h\ op\ ?v}\ (\text{AddCol})$$

Fig. 10. Repair tactics. Here, *split*(v) tokenizes value v using predefined delimiters: *split*(v) = (v_1, v_2) iff v_1 occurs before the first occurrence of the delimiter and v_2 occurs after. If the delimiter doesn't appear in v, then *split*(v) = (v, ϵ).

faulty sub-sketch $\mathcal{F}$ has the property that all of its strict sub-expressions have an inhabitant whose score exceeds our confidence threshold, but $\mathcal{F}$ itself does not.

Algorithm 2 presents our fault localization algorithm for query sketches. The recursive FaultLocalize procedure takes as input the database schema Γ, a partial sketch $\mathcal{S}$, which can be either a relation or a specifier. If $\mathcal{S}$ is a specifier, FaultLocalize also takes as input a record type τ, which describes the schema for the parent table. (Recall that sketch completion for specifiers requires the parent table τ.) The return value of FaultLocalize is either the faulty sub-sketch or null (if $\mathcal{S}$ cannot be repaired).

Let us now consider Algorithm 2 in more detail. If $\mathcal{S}$ is a relation, we first recurse down to its subrelations and specifiers (see Figure 9) to identify a smaller subterm that can be repaired (lines 4–13). On the other hand, if $\mathcal{S}$ is a specifier (lines 14–17), 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 $\mathcal{S}$ as the possible cause of failure. That is, if $\mathcal{S}$ does not have any inhabitants that meet our confidence threshold, we then check if $\mathcal{S}$ can be repaired using our database of repair tactics. If so, we return $\mathcal{S}$ as the problematic subterm (lines 18–20).

One subtle part of the fault localization procedure is the handling of specifiers in lines 11–13. Recall from Section 6 that the completion of specifiers is dependent on the parent relation. Specifically, when we find the inhabitant of a relation such as $\Pi_\kappa(\chi)$, we need to know the type of χ when we complete κ. Hence, there is a valid completion of $\Pi_\kappa(\chi)$ if there *exists* a valid completion of κ for *some* inhabitant of χ. Thus, we can only say that ω_i (or one of its subterms) is faulty if it is faulty for all possible inhabitants of χ (i.e., $\forall j.\ \omega'_{ij} \neq$ null).

Example 7.1. Consider the query "Find the number of papers in OOPSLA 2010" from the motivating example in Section 2. The initial sketch generated by semantic parsing is:

$$\Pi_{count(?papers)}(\sigma_{?=OOPSLA\ 2010}(??papers))$$

Since there is no high-confidence completion of this sketch, we perform fault localization using Algorithm 2. The innermost hole `??papers` can be instantiated with high confidence, so we next consider the subterm `? = OOPSLA 2010`. Observe that there is no completion of `??papers` under which `? = OOPSLA 2010` has a high confidence score because no table in the database contains the entry `"OOPSLA 2010"`. Hence, fault localization identifies the predicate as `? = OOPSLA 2010` as the root cause of failure.

Once we identify a faulty subpart $\mathcal{F}$ of sketch $\mathcal{S}$, our method tries to repair $\mathcal{S}$ by replacing $\mathcal{F}$ by some $\mathcal{F}'$ obtained using a database of domain-specific repair tactics. Figure 10 shows a representative subset of our repair tactics for the query synthesis domain. At a high-level, our repair tactics describe how to introduce new predicates, join operators, and columns into the relevant sub-parts of the sketch.

To gain some intuition about our repair tactics for database queries, let us consider the rewrite rules in Figure 10 in more detail. The first tactic, labeled *AddPred*, splits a predicate into two parts by introducing a conjunct. For instance, consider a predicate $?h\ op\ v$ and suppose that v is a string that contains a common delimiter (e.g., space, apostrophe etc.). In this case, the *AddPred* tactic splits v into two parts v_1, v_2 occurring before and after the delimiter and rewrites the predicate as $?h\ op\ v_1 \wedge ?h\ op\ v_2$. For example, we have used this tactic in the motivating example from Section 2 when rewriting `?="OOPSLA 2010"` as `?="OOPSLA"` and `?="2010"`.

The rewrite rules labeled *AddJoin* in Figure 10 show how to introduce additional join operators in selections, projections, and joins. Because users may not be aware that the relevant information is spread across multiple database tables, these tactics allow us to introduce join operators when the user's English description does not contain any clues about the necessity of performing joins. For instance, recall that we use the *AddJoin* tactic in our example from Section 2 to rewrite the term `??[papers]` into `??[papers] JOIN ??`.

The next rule labeled *AddFunc* introduces an aggregate function if the hint h in $?h$ contains the name of an aggregate function (e.g., count) or something similar to it. For instance, consider a hole `?` with hint *"average grade"*. Since *"average"* is also an aggregate function, the *AddFunc* rule can be used to rewrite this as *avg(?grade)*. Finally, the last rule labeled *AddCol* introduces a new column in the predicate $?h\ op\ v$. Since v may refer to the name of a column rather than a constant string value, the *AddCol* rule allows us to consider this alternative possibility.

As mentioned earlier, the particular repair tactics used in the context of sketch refinement are quite domain-specific. However, since natural language is inherently imprecise and ambiguous, we believe that the proposed methodology of refining the program sketch using fault localization and repair tactics would also be beneficial in other contexts where the goal is to synthesize a program from natural language.

8 IMPLEMENTATION

Our Sqlizer tool, written in a combination of C++ and Java, automatically synthesizes SQL code from English queries. Sqlizer uses the Sempre framework [Berant et al. 2013] and the Stanford CoreNLP library [Manning et al. 2014] in the implementation of its semantic parser. For quantitative type inhabitation, Sqlizer uses the Word2Vec [Mikolov et al. 2013] tool to compute a similarity metric between English hints in the sketch and names of database tables and columns.

Recall that our synthesis algorithm presented in Section 3 only considers the top k query sketches and repairs each program sketch at most n times. It also uses a threshold γ to reject low-confidence queries. While each of these parameters can be configured by the user, the default values for k and n are both 5, and the default value for γ is 0.35. In our experimental evaluation, we also use these default values.

Training data. Recall from Section 5 that our semantic parser uses supervised machine learning to optimize the weights used in the likelihood score for each utterance. Towards this goal, we used queries for a mock database accompanying a databases textbook [Elmasri and Navathe 2011]. Specifically, this database contains information about the employees, departments, and projects for a hypothetical company. In order to train the semantic parser, we extracted the English descriptions of 108 queries from the textbook and manually wrote the corresponding sketch for each query. Please note that the query sketches were constructed directly from the English description *without manually "repairing" them to fit the actual database schema.* Also, while a training set of 108 queries may seem like a small number compared to other supervised machine learning techniques, we can get away with such a modest amount of training data for several reasons: First, due to our use of query sketches as the logical form representation, we do not need to train a statistical model to map English phrases to relational algebra expressions over the database schema. Second, the grammar we implemented in the semantic parser is carefully designed to minimize ambiguities. Finally, the linguistic processor used in the semantic parser is pre-trained over a large corpus, including ~2,500 articles from the Wall Street Journal for part-of-speech tagging and ~15,000 sentences from the CoNLL-2003 dataset for named entity recognition.

Optimizations. While our implementation closely follows the technical presentation in this paper, it performs two important optimizations that we have not mentioned previously: First, because the fault localization procedure completes the same sub-sketch many times, we memoize the result of sketch completion for every subterm. Second, if the score for a subterm is less than a certain confidence threshold, we reject the partially completed sketch without trying to complete the remaining holes in the sketch.

9 EVALUATION

To evaluate Sqlizer, we perform experiments that are designed to answer the following questions:

Q1. How effective is Sqlizer at synthesizing SQL queries from natural language descriptions?
Q2. What is Sqlizer's average running time per query?
Q3. How well does Sqlizer perform across different databases?
Q4. How does Sqlizer perform compared to other tools for synthesizing queries from natural language?
Q5. What is the relative importance of type information and sketch repair in practice?
Q6. How important are the various heuristics that we use to assign confidence scores to type inhabitants?

9.1 Experimental Setup

To answer these research questions, we evaluate Sqlizer on three real-world databases, namely the Microsoft academic search database (MAS) used for evaluating Nalir [Li and Jagadish 2014], the Yelp business reviewing database (YELP), and the IMDB movie database (IMDB). [4] Table 1 provides statistics about each database.

Benchmarks. To evaluate Sqlizer on these databases, we collected a total of 455 natural language queries. For the MAS database, we use exactly the 196 benchmarks obtained from the Nalir dataset [Li and Jagadish 2014]. For the IMDB and YELP databases, we asked a group of people at our organization to come up with English queries that they might like to answer using IMDB and YELP. Participants were given information about the type of data available in each database

[4] All the benchmarks and databases are available at goo.gl/DbUBMM

Table 1. Database Statistics

Database	Size	#Tables	#Columns
MAS	3.2GB	17	53
IMDB	1.3GB	16	65
YELP	2.0GB	7	38

Table 2. Categorization of different benchmarks

Cat	Description
C1	Does not use aggregate function or join operator
C2	Does not use aggregate function but Joins different tables
C3	Uses an aggregate function
C4	Uses subquery or self-join

(e.g., business names, cities, etc.), but they did not have any prior knowledge about the underlying database schema, including names of database tables and columns.

Checking correctness. To evaluate the accuracy of Sqlizer, we manually inspected the SQL queries returned by Sqlizer. We consider a query to be correct if (a) executing the query yields the desired information, and (b) the synthesized query faithfully implements the data retrieval logic specified by the user's English description.

Categorization of benchmarks. To assess how Sqlizer performs on different classes of queries, we manually categorize the benchmarks into four groups based on the characteristics of their corresponding SQL query. Table 2 shows our taxonomy and provides an English description for each category. While there is no universal agreement on the difficulty level of a given database query, we believe that benchmarks in category C_{i+1} are generally harder for humans to write than benchmarks in category C_i.

Hardware and OS. All of our experiments are conducted on an Intel Xeon(R) computer with E5-1620 v3 CPU and 32GB memory, running the Ubuntu 14.04 operating system.

9.2 Accuracy and Running Time

Table 3 summarizes the results of evaluating Sqlizer on the 455 benchmarks involving the MAS, IMDB, and YELP databases. In this table, the column labeled "Count" shows the number of benchmarks under each category. The columns labeled "Top k" show the number (#) and percentage (%) of benchmarks whose target query is ranked within the top k queries synthesized by Sqlizer. Finally, the columns labeled "Parse time" and "Synth/repair time" show the average time (in seconds) for semantic parsing and sketch completion/refinement respectively.

As shown in Table 3, Sqlizer achieves close to 90% accuracy across all three databases when we consider a benchmark to be successful if the desired query appears within the top 5 results. Even if we adopt a stricter definition of success and consider Sqlizer to be successful if the target query is ranked within the top one (resp. top three) results, Sqlizer still achieves approximately 78% (resp. 86%) accuracy. Also, observe that Sqlizer's synthesis time is quite reasonable; on average, Sqlizer takes 1.22 seconds to synthesize each query, with 85% of synthesis time dominated by semantic parsing.

To gain intuition about cases in which Sqlizer does not work well, we investigated the root causes of failures in our experimental evaluation. In most cases, the problem is caused by domain-specific terms for which we cannot accurately compute similarities using Word2Vec. For instance, in the context of on-line reviewing systems such as Yelp, the terms "star" and "rating" are used interchangeably, but the domain-agnostic Word2Vec system does not consider these two terms to be similar. Clearly, this problem can be alleviated by training the neural net for measuring

Table 3. Summary of our experimental evaluation

DB	Cat	Count	Top 1		Top 3		Top 5		Parse time (s)	Synth/repair time (s)
			#	%	#	%	#	%		
MAS	C1	14	12	85.7	14	100.0	14	100.0	0.41	0.08
	C2	59	52	88.1	55	93.2	55	93.2	1.07	0.16
	C3	60	49	81.7	55	91.6	56	93.3	1.11	0.29
	C4	63	45	71.4	49	77.7	53	84.1	2.53	0.21
	Total	**196**	**158**	**80.6**	**173**	**88.3**	**178**	**90.8**	**1.50**	**0.21**
IMDB	C1	18	16	88.9	17	94.4	17	94.4	0.50	0.21
	C2	69	51	73.9	59	85.5	61	88.4	0.60	0.24
	C3	27	24	88.8	26	96.2	26	96.2	0.71	0.34
	C4	17	11	64.7	11	64.7	12	70.5	0.70	0.49
	Total	**131**	**102**	**77.9**	**113**	**86.3**	**116**	**88.5**	**0.61**	**0.28**
YELP	C1	8	6	75.0	7	87.5	7	87.5	0.55	0.02
	C2	49	35	71.4	39	79.6	42	85.7	0.77	0.05
	C3	51	40	78.4	48	94.1	49	96.0	0.72	0.05
	C4	20	15	75.0	15	75.0	15	75.0	0.96	0.06
	Total	**128**	**96**	**75.0**	**109**	**85.2**	**113**	**88.3**	**0.77**	**0.05**

word similarity on a corpus specialized for this domain. However, since our goal is to develop a database-agnostic system, we have not performed such domain-specific training for our evaluation.

9.3 Comparison with NALIR

To evaluate whether the results described in Section 9.2 improve over the state-of-the-art, we also compare Sqlizer against Nalir, a recent system that won a best paper award at VLDB'14 [Li and Jagadish 2014]. Rather than re-implementing the ideas proposed in the VLDB'14 paper, we directly use the Nalir implementation provided to us by Nalir's developers.

Similar to Sqlizer, Nalir generates SQL queries from English and also aims to be database-agnostic (i.e., does not require database-specific training). However, unlike Sqlizer, which is fully automated, Nalir can also be used in an interactive setting that allows the user to provide guidance by choosing the right query structure or the names of database elements. In order to perform a fair comparison between Sqlizer and Nalir, we use Nalir in the non-interactive setting (recall that Sqlizer is fully automatic). Furthermore, since Nalir only generates a single database query as its output, we compare Nalir's results with the *top-ranked query* produced by Sqlizer.

As shown in Figure 11, Sqlizer outperforms Nalir on all three databases with respect to the number of benchmarks that can be solved. In particular, Sqlizer's average accuracy is 78% whereas Nalir's average accuracy is less than 32%. Observe that the queries for the MAS database are the same ones used for evaluating Nalir, but Sqlizer outperforms Nalir even on this dataset. Furthermore, even though Sqlizer's accuracy is roughly the same across all three databases, Nalir performs significantly worse on the IMDB and YELP databases.

To provide some intuition about why Sqlizer performs better than Nalir in our experiments, recall that Sqlizer can automatically refine query sketches using a database of repair tactics and guided by the confidence scores inferred during sketch completion. In contrast, Nalir does not automatically resolve ambiguities and is more effective when it is used in its interactive mode that

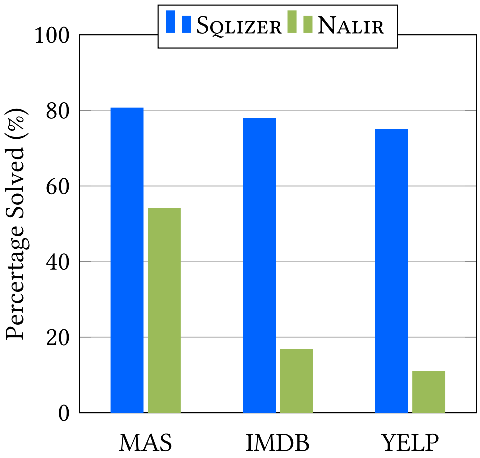

Fig. 11. Comparison between SQLIZER and NALIR

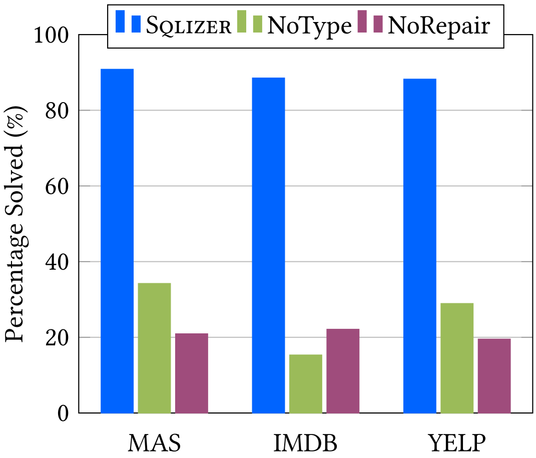

Fig. 12. Comparison between different variations of SQLIZER

allows the user to guide the system by choosing between different candidate mappings between nodes in the parse tree to SQL components.

9.4 Evaluation of Different Components of Synthesis Methodology

In this paper, we argued that the use type information and automatic sketch refinement are both very important for effective synthesis from natural language. To justify this argument, we compare SQLIZER against two variants of itself. One variant, referred to as **NoType**, does not use type information to reject some of the generated SQL queries. The second variant, referred to as **NoRepair**, does not perform sketch refinement.

Figure 12 shows the results of our evaluation comparing SQLIZER against these two variants of itself. As we can see from this figure, disabling either of these features dramatically reduces the accuracy of the system. In particular, while the full SQLIZER system ranks the target query within the top 5 results in over 88% of the cases, the average accuracy of both variants is below 35%. We believe that these results demonstrate that the use of types and automated repair are both crucial for the overall effectiveness of SQLIZER.

9.5 Evaluation of Heuristics for Assigning Confidence Scores

A key ingredient of the synthesis methodology proposed in this paper is *quantitative type inhabitation* in which we use domain-specific heuristics to assign confidence scores to programs. In the context of the database domain, we proposed three different heuristics for assigning confidence scores to queries. The first heuristic, referred to as *Hints*, computes the similarity between the natural language hints in the sketch and the names of schema elements. The second heuristic, referred to as *Keys*, uses a function $P_{\bowtie}$ to assign scores to join operators using information about foreign keys. The third heuristic, referred to as *Content*, uses a function P_{ϕ} that assigns confidence scores to selection operations using the contents of the database.

We evaluate the relative importance of each of these heuristics in Figure 13. Specifically, the variant of SQLIZER labeled *NoX* disables the scoring heuristic called *X*. As we can see from the figure, all of our proposed heuristics are quite important for SQLIZER to work effectively as an end-to-end synthesis tool.

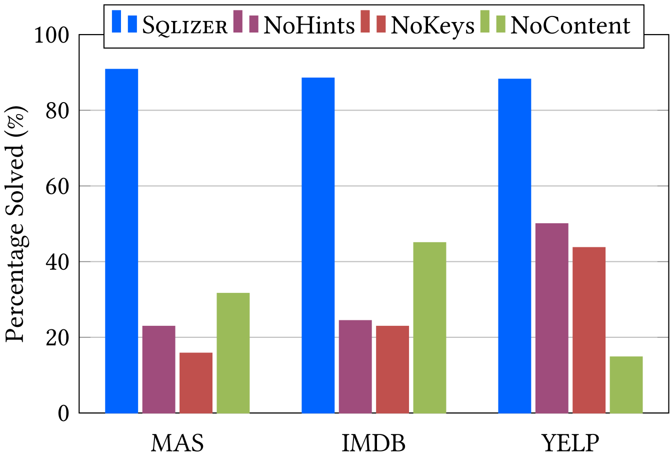

Fig. 13. Impact of heuristics for assigning confidence scores on Top 5 results

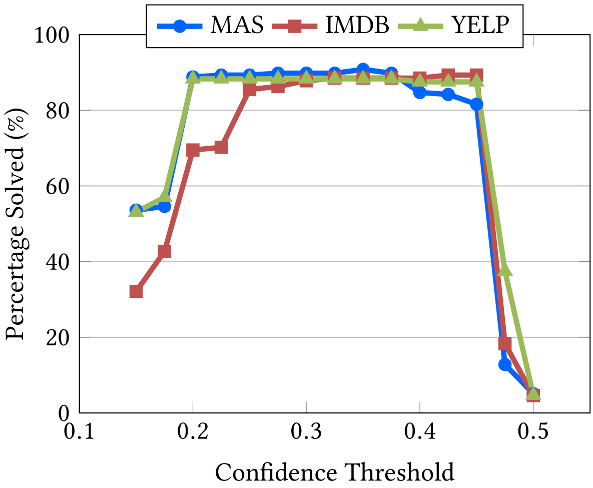

Fig. 14. Impact of different confidence thresholds on Top 5 results

9.6 Evaluation of Different Confidence Thresholds

Recall from Algorithm 1 that our synthesis algorithm rejects queries with a confidence score below a certain threshold γ. As mentioned in Section 8, we use the default value $\gamma = 0.35$ in all of our experiments. To understand the impact of this confidence threshold on our results, we also run the same set of experiments using different confidence thresholds in the range $[0.15, 0.5]$. As shown in Figure 14, SQLIZER is not very sensitive to the exact value of γ as long as it is in the range $[0.25, 0.45]$; however, accuracy drops sharply if γ is chosen to be either below 0.25 or above 0.45. If the threshold γ is too low (i.e., below 0.25), SQLIZER seems to generate many incorrect queries, thereby affecting the overall ranking of the target query. On the other hand, if γ is too high (i.e., above 0.45), SQLIZER ends up ruling out more queries, sometimes including the desired query. Hence, overall precision decreases in both cases.

10 RELATED WORK

The work presented in this paper spans a variety of different topics from the databases, programming languages, and natural language processing communities. In what follows, we compare our technique with related approaches.

Database query synthesis. There is a significant body of work on automatically synthesizing database queries. Related work in this area can be categorized into three classes, depending on the form of specifications provided by the user. In one line of work on query synthesis, users convey their intent to the system through the use of *input-output examples* [Tran et al. 2009; Wang et al. 2017; Zhang and Sun 2013; Zloof 1975]. Specifically, the input to the system is a miniature version of the database, and the output is the desired table that should be extracted from this database using the target query. In this work, we prefer natural language specifications over input-output examples for two important reasons: First, in order to provide input-output examples, the user must be familiar with the database schema, which is not always the case. Second, since a database may contain several tables with many different columns, providing input-output examples may be prohibitively cumbersome.

The second line of work on query synthesis uses *natural language descriptions* to convey user intent [Androutsopoulos et al. 1993, 1995; Li and Jagadish 2014; Li et al. 2006; Popescu et al. 2004, 2003; Warren and Pereira 1982]. Early work in this area focuses on systems that are hand-crafted to specific databases [Codd 1974; Hendrix et al. 1978; Warren and Pereira 1982; Woods et al. 1972]. Later work describes NLIDB systems that can be reused for multiple databases with appropriate customization [Grosz et al. 1987; Tang and Mooney 2000; Zelle and Mooney 1996]. However, these techniques are not database-agnostic in that they require additional customization for each database. In contrast, our technique does not require database-specific training.

Similar to our proposed approach, the Nalir system [Li and Jagadish 2014] also aims to be database-agnostic. Specifically, Nalir leverages an English dependency parser to generate linguistic parse trees, which are subsequently translated into *query trees*, possibly with guidance from the user. In addition to being relatively easy to convert to SQL, these query trees can also be translated back into natural language with the goal of facilitating user interaction. In contrast, our goal in this paper is to develop a system that is as reliable as Nalir without requiring guidance from the user. In particular, since users may not be familiar with the underlying database schema, it may be difficult for them to answer some of the questions posed by a Nalir-style system. In contrast, our approach does not assume that users are familiar with the organization of information in the database.

The third line of work on query synthesis generates more efficient SQL queries from code written in conventional programming languages [Cheung et al. 2013; Wiedermann et al. 2008]. For instance, QBS [Cheung et al. 2013] transforms parts of the application logic into SQL queries by automatically inferring loop invariants. This line of work is not tailored towards end-users and can be viewed as a form of query optimization using static analysis of source code.

Programming by natural language. Since SQL is a declarative language, the techniques proposed in this paper are also related to *programming by natural language* [Desai et al. 2016; Gulwani and Marron 2014; Le et al. 2013; Quirk et al. 2015; Raza et al. 2015]. In addition to query synthesis, natural language has also been used as the preferred specification mechanism in the context of smartphone automation scripts [Le et al. 2013], “if-then-else recipes" [Quirk et al. 2015], spreadsheet programming [Gulwani and Marron 2014], and string manipulation [Raza et al. 2015]. Among these systems, the NLyze tool [Gulwani and Marron 2014] is most closely related to Sqlizer in that it also combines semantic parsing with type-directed synthesis. However, NLyze does not generate program sketches and uses type-based synthesis to mitigate the low recall of semantic parsing. In contrast, Sqlizer uses semantic parsing to generate an initial query sketch, which is refined using repair tactics and completed using quantitative type inhabitation. Furthermore, NLyze targets spreadsheets rather than relational databases and proposes a new DSL for this domain.

Most recently, [Desai et al. 2016] have proposed a general framework for constructing synthesizers that can generate programs in a domain-specific DSL from English descriptions. The framework requires a DSL definition and a set of domain-specific training data in the form of pairs of English sentences and their corresponding programs in the given DSL. While this framework can, in principle, be instantiated for SQL query synthesis, it would require database-specific training data.

Program synthesis. The techniques proposed in this paper borrow insights from other papers on program synthesis. In particular, the use of the term *sketch* is inspired by the Sketch system [Solar Lezama 2008; Solar-Lezama et al. 2005, 2006] in which the user writes a *program sketch* containing holes (unknown expressions). However, in contrast to the Sketch system where the holes are instantiated with constants, holes in our query sketches are completed using tables, columns, and predicates. Similar to Sqlizer, some prior techniques (e.g., [Feng et al. 2017b; Zhang and Sun 2013]) have also decomposed the synthesis task into two separate sketch generation and sketch completion phases. However, to the best of our knowledge, we are the first to generate program sketches from natural language using semantic parsing.

In addition to program sketching, this paper also draws insights from recent work on *type-directed program synthesis* [Feser et al. 2015; Gvero et al. 2013; Osera and Zdancewic 2015; Polikarpova et al. 2016]. Among these projects, the most related one is the InSynth system, which synthesizes small code snippets in a type-directed manner. Similar to our proposed methodology, InSynth also performs quantitative type inhabitation to synthesize type-correct expressions at a given program point. Specifically, InSynth assigns weights to each type inhabitant, where lower weights indicate higher relevance of the synthesized term. These weights are derived using a training corpus and the structure of the code snippet. At a high level, our use of quantitative type inhabitation is similar to InSynth in that we both use numerical scores to evaluate which term is most likely to be the inhabitant desired by the user. However, the way in which we assign confidence scores to type inhabitants is very different from the way InSynth assigns weights to synthesized code snippets. Furthermore, in contrast to InSynth where lower weights indicate higher relevance, Sqlizer assigns lower scores to inhabitants that are less likely to be correct.

Fault localization and program repair. As mentioned earlier, the sketch refinement strategy in Sqlizer is inspired by prior work on *fault localization* [Ball et al. 2003; Groce and Visser 2003; Jones and Harrold 2005; Jones et al. 2002; Jose and Majumdar 2011a,b] and *program repair* [Goues et al. 2012; Long and Rinard 2015, 2016; Nguyen et al. 2013; Weimer et al. 2009]. Similar to other techniques on program repair, the repair tactics employed by Sqlizer can be viewed as a pre-defined set of templates for mutating the program. However, to the best of our knowledge, we are the first to apply repair at the level of program sketches rather than programs.

Fault localization techniques [Ball et al. 2003; Groce and Visser 2003; Jones and Harrold 2005; Jones et al. 2002; Jose and Majumdar 2011a,b] aim to pinpoint a program expression that corresponds to the root cause of a bug. Similar to these fault localization techniques, Sqlizer tries to identify the minimal faulty subpart of the “program". However, we perform fault localization at the level of program sketches by finding a minimal sub-sketch for which no high-confidence completion exists.

Semantic parsing. Unlike syntactic parsing which focuses on the grammatical divisions of a sentence, semantic parsing aims to represent a sentence through logical forms expressed in some formal language [Berant et al. 2013; Kate et al. 2005; Liang and Potts 2015; Tang and Mooney 2000;

Zelle and Mooney 1993]. Previous techniques have used semantic parsing to directly translate English sentences to queries [Kate et al. 2005; Miller et al. 1996; Tang and Mooney 2000; Zelle and Mooney 1996]. Unlike these techniques, we only use semantic parsing to generate an initial query sketch rather than the full query. We believe that there are two key advantages to our approach: First, our technique can be used to answer queries on a database on which it has not been previously trained. Second, the use of sketch refinement allows us to handle situations where the user's description does not accurately reflect the underlying database schema.

11 LIMITATIONS AND FUTURE WORK

In this section, we discuss the current limitations of our system and possible ways to improve it in the future.

Recall that Sqlizer uses domain-specific heuristics to assign confidence scores to query completions, and one of these heuristics (namely, P_ϕ) uses database contents when performing quantitative type inhabitation. The strategy of assigning low confidence scores to queries that yield an empty relation works very well in most cases, but it may prevent Sqlizer from generating the right query if the query legitimately returns an empty table. Another situation in which this heuristic may not work well is if the user's natural language query does not *exactly* match the contents of the database, such as in cases where the user's description uses an abbreviation or contains a misspelling. A possible solution to this problem is to look for syntactically or semantically similar entries in the database rather than insisting on an exact match.

Another limitation of Sqlizer is that the user ultimately needs to decide which (if any) of the top k queries returned by Sqlizer is the right one. An interesting avenue for future work is to explore user-friendly mechanisms to help the user make this decision. One possibility is to present the user with a natural language description of the query rather than the SQL query itself. However, this solution would still require the user to be knowledgeable about the underlying database schema. Another possibility is to present the query *result* rather than the query itself and let the user decide if the result is sensible. An even better solution would be to further improve the system's accuracy so that the desired SQL query is ranked number one in more cases. However, given that Sqlizer is already capable of returning the desired query as the top result in 78% of the cases, we believe that Sqlizer is still quite useful to end-users as is.

12 CONCLUSIONS

We have proposed a new methodology for synthesizing programs from natural language and applied it to the problem of synthesizing SQL code from English queries. Starting with an initial program sketch generated using semantic parsing, our approach enters an iterative refinement loop that alternates between quantitative type inhabitation and sketch repair. Specifically, our method uses domain-specific knowledge to assign confidence scores to type (i.e., sketch) inhabitants and uses these confidence scores to guide fault localization. The faulty subterms pinpointed using error localization are then repaired using a database of domain-specific repair tactics.

We have implemented the proposed approach in a tool called Sqlizer, an end-to-end system for generating SQL queries from natural language. Our experiments on 455 queries from three different databases shows that Sqlizer ranks the desired query as top one in 78% of the cases and among top 5 in $\sim$ 90% of the time. Our experiments also show that Sqlizer significantly outperforms Nalir, a state-of-the-art system for generating SQL code from natural language queries. Finally, our evaluation also justifies the importance of type information and program repair and shows that our proposed domain-specific heuristics are necessary and synergistic.

In the future, we plan to apply our proposed synthesis methodology to other domains where it is beneficial to generate code from English descriptions. For instance, we believe that our proposed synthesis methodology could be also useful for querying data stored in other forms (e.g., noSQL databases, XML documents, file systems) or for synthesizing simple scripts, such as if-then-else recipes or robot control commands.

ACKNOWLEDGMENTS

We would like to thank Xinyu Wang and the anonymous reviewers for their thorough and helpful comments. This material is based on research sponsored by DARPA under agreement number #8750-14-2-0270. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

I Androutsopoulos, G Ritchie, and P Thanisch. 1993. Masque/sql: An Efficient and Portable Natural Language Query Interface for Relational Databases. *Tech report, University of Edinburgh* (1993).

Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. 1995. Natural language interfaces to databases - An Introduction. *Natural Language Engineering* (1995).

Thomas Ball, Mayur Naik, and Sriram K. Rajamani. 2003. From symptom to cause: localizing errors in counterexample traces. In *POPL*. 97–105.

Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin G. Zorn. 2015. FlashRelate: extracting relational data from semi-structured spreadsheets using examples. ACM, 218–228.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic Parsing on Freebase from Question-Answer Pairs. In *EMNLP*. 1533–1544.

Bob Carpenter. 1997. *Type-logical semantics.* MIT press.

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *PLDI*. 3–14.

E. F. Codd. 1974. Seven Steps to Rendezvous with the Casual User. In *IFIP Working Conference Data Base Management*. 179–200.

Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. 2016. Program synthesis using natural language. In *ICSE*.

Ramez Elmasri and Shamkant B. Navathe. 2011. *Fundamentals of Database Systems.* Addison-Wesley.

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017a. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Programming Language Design and Implementation*.

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017b. Component-based synthesis for complex APIs. In *POPL*. 599–612.

John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *PLDI*. 229–239.

Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair. In *ICSE*. 3–13.

Alex Groce and Willem Visser. 2003. What Went Wrong: Explaining Counterexamples. In *SPIN*.

Barbara J. Grosz, Douglas E. Appelt, Paul A. Martin, and Fernando C. N. Pereira. 1987. TEAM: An Experiment in the Design of Transportable Natural-Language Interfaces. *Artificial Intelligence* 32, 2 (1987), 173–243.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 317–330.

Sumit Gulwani and Mark Marron. 2014. NLyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In *SIGMOD*.

Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 27–38.

Gary G. Hendrix, Earl D. Sacerdoti, Daniel Sagalowicz, and Jonathan Slocum. 1978. Developing a Natural Language Interface to Complex Data. *TODS* 3, 2 (1978), 105–147.

James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*. 273–282.

James A. Jones, Mary Jean Harrold, and John T. Stasko. 2002. Visualization of test information to assist fault localization. In *ICSE*. 467–477.

Manu Jose and Rupak Majumdar. 2011a. Bug-Assist: assisting fault localization in ANSI-C programs. In *CAV*.

Manu Jose and Rupak Majumdar. 2011b. Cause clue clauses: error localization using maximum satisfiability. *PLDI* (2011).

Rohit J. Kate and Raymond J. Mooney. 2006. Using String-Kernels for Learning Semantic Parsers. In *ACL*.

Rohit J. Kate, Yuk Wah Wong, and Raymond J. Mooney. 2005. Learning to Transform Natural to Formal Languages. In *AAAI*. 1062–1068.

Vu Le and Sumit Gulwani. 2014. FlashExtract: a framework for data extraction by examples. ACM, 542–553.

Vu Le, Sumit Gulwani, and Zhendong Su. 2013. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *MobiSys*. 193–206.

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

Yunyao Li, Huahai Yang, and H. V. Jagadish. 2006. Constructing a Generic Natural Language Interface for an XML Database. In *EDBT*. 737–754.

Percy Liang and Christopher Potts. 2015. Bringing machine learning and compositional semantics together. *Annual Review of Linguistics* 1, 1 (2015), 355–376.

Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *ESEC/FSE*.

Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *POPL*.

Bill MacCartney and Christopher D Manning. 2009. An extended model of natural logic. In *IWCS*. 140–156.

Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *ACL System Demonstrations*. 55–60.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *NIPS*.

Scott Miller, David Stallard, Robert J. Bobrow, and Richard M. Schwartz. 1996. A Fully Statistical Approach to Natural Language Interfaces. In *ACL*. 55–61.

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *ICSE*. 772–781.

Peter-Michael Osera and Steve Zdancewic. 2015. Type- and example-directed program synthesis. In *PLDI*.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *PLDI*. 522–538.

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A framework for inductive program synthesis. ACM, 107–126.

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*.

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

Chris Quirk, Raymond Mooney, and Michel Galley. 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. In *ACL*. 878–888.

Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional Program Synthesis from Natural Language and Examples. In *IJCAI*.

Armando Solar Lezama. 2008. *Program Synthesis By Sketching*. Ph.D. Dissertation. EECS Department, University of California, Berkeley.

Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. 2005. Programming by sketching for bit-streaming programs. In *PLDI*.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *ASPLOS*. 404–415.

Lappoon R Tang and Raymond J Mooney. 2000. Automated construction of database interfaces: Integrating statistical and relational learning for semantic parsing. In *EMNLP*. 133–141.

Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. 2009. Query by output. In *SIGMOD*. 535–548.

Chenglong Wang, Alvin Cheung, and Ras Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. In *Programming Language Design and Implementation*.

David H. D. Warren and Fernando C. N. Pereira. 1982. An Efficient Easily Adaptable System for Interpreting Natural Language Queries. *American Journal of Computational Linguistics* 8, 3-4 (1982), 110–122.

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *ICSE*. 364–374.

Ben Wiedermann, Ali Ibrahim, and William R. Cook. 2008. Interprocedural query extraction for transparent persistence. In *OOPSLA*. 19–36.

William A Woods, Ronald M Kaplan, and Bonnie Nash-Webber. 1972. *The lunar sciences natural language information system*. Bolt, Beranek and Newman.

Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing transformations on hierarchically structured data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and*

Implementation. ACM, 508–521.

John M. Zelle and Raymond J. Mooney. 1993. Learning Semantic Grammars with Constructive Inductive Logic Programming. In *AAAI*. 817–822.

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

Sai Zhang and Yuyin Sun. 2013. Automatically synthesizing SQL queries from input-output examples. In *ASE*.

Moshé M. Zloof. 1975. Query-by-Example: the Invocation and Definition of Tables and Forms. In *VLDB*.