Duoquest: A Dual-Specification System for Expressive SQL Queries

Christopher Baik
cjbaik@umich.edu
University of Michigan
Ann Arbor, MI, USA

Zhongjun Jin
markjin@umich.edu
University of Michigan
Ann Arbor, MI, USA

Michael Cafarella
michjc@umich.edu
University of Michigan
Ann Arbor, MI, USA

H. V. Jagadish
jag@umich.edu
University of Michigan
Ann Arbor, MI, USA

ABSTRACT

Querying a relational database is difficult because it requires users to be familiar with both the SQL language and the schema. However, many users possess enough domain expertise to describe their desired queries by alternative means. For such users, two major alternatives to writing SQL are natural language interfaces (NLIs) and programming-by-example (PBE). Both of these alternatives face certain pitfalls: natural language queries (NLQs) are often ambiguous, even for human interpreters, while current PBE approaches limit functionality to be tractable. Consequently, we propose *dual-specification query synthesis*, which consumes both a NLQ and an optional PBE-like *table sketch query* that enables users to express varied levels of domain knowledge. We introduce the novel dual-specification Duoquest system, which leverages *guided partial query enumeration* to efficiently explore the space of possible queries. We present results from user studies in which Duoquest demonstrates a 62.5% absolute increase in query construction accuracy over a state-of-the-art NLI and comparable accuracy to a PBE system on a limited workload supported by the PBE system. In a simulation study on the Spider benchmark, Duoquest demonstrates a >2x increase in top-1 accuracy over both NLI and PBE.

ACM Reference Format:
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 (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3318464.3389776

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

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389776

1 INTRODUCTION

Querying a relational database is difficult because it requires users to know both the SQL language and be familiar with the schema. On the other hand, many users possess enough domain familiarity or expertise to describe their desired queries by alternative means. Consequently, an ongoing research challenge is enabling users with domain-specific knowledge but little to no programming background to specify queries.

One popular approach is the natural language interface (NLI), where users can state queries in their native language. Unfortunately, existing NLIs require significant overhead in adapting to new domains and databases [12, 15, 22] or are overly reliant on specific sentence structures [8]. More recent advances leverage deep learning in an attempt to circumvent these challenges, but the state-of-the-art accuracy [23] on established benchmarks falls well short of the desired outcome, which is that NLIs should either interpret the user's query correctly or clearly detect any errors [12].

Another alternative to writing SQL is programming-by-example (PBE), where users must either provide query output examples or example pairs of an input database and the output of the desired query. PBE systems have the advantage of a concrete notion of *soundness* in that returned candidate queries are guaranteed to satisfy the user's specification, while NLIs, on the other hand, provide no such guarantees.

However, PBE systems must precariously juggle various factors: how much *query expressiveness* is permitted, whether *schema knowledge* is required of the user, whether users may provide *partial tuples* rather than full tuples, and whether an *open- or closed-world setting* is assumed, where in a closed-world setting, the user is expected to provide a complete result set, while the user may provide a subset of possible returned tuples in an open-world setting.

Table 1 summarizes the capabilities of previous NLI and PBE systems, with respect to three major categories:

(1) *soundness*, which guarantees that results satisfy the user specification;
(2) permitted *query expressiveness*;
(3) and required *user knowledge*.

With respect to these factors, an ideal system would: (1) provide soundness guarantees; (2) enable expressive queries

System	Soundness	Query Expr.[1] ⋈	σ	γ	Knowledge[2] NS	PT	OW
NLIs [8, 22, 23]		✓	✓	✓	✓	N/A	N/A
PBE Systems							
QBE [25]	✓	✓	✓	✓		✓	✓
MWeaver [14]	✓	✓			✓		✓
S4 [13]	✓	✓			✓	✓	✓
SQuID [6]	✓	✓	✓	✓[3]	✓		✓
TALOS [18]	✓	✓	✓	✓	✓		
QFE [9]	✓	✓	✓				
PALEO [11]	✓		✓	✓			
Scythe [19]	✓	✓	✓	✓			
REGAL+ [17]	✓	✓	✓	✓	✓		
DUOQUEST	✓	✓	✓	✓	✓	✓	✓

Table 1: DUOQUEST vs. NLI/PBE, considering soundness, query expressiveness, and required user knowledge. A ✓ is desirable in each column.

with selections, aggregates, and joins; and (3) allow users to provide partial tuples in an open-world setting without schema knowledge. However, previous approaches could not handle the massive search space produced by this scenario and each constrained at least one of the above factors.

Our Approach — While existing approaches only permit users to specify a single specification, we observe that PBE specifications and natural language queries (NLQs) are complementary, as PBE specifications contain hard constraints that can substantially prune the search space, while NLQs provide hints on the structure of the desired SQL query, such as selection predicates and the presence of clauses. Therefore, we argue for *dual-specification query synthesis*, which consumes both a NLQ and an optional PBE-like specification as input. The dual-specification approach does not inhibit users who are only able to provide a single specification, but can help the system more easily triangulate the desired query when users are able to provide both types of specifications.

System Desiderata — There are several goals in developing a dual-specification system.

First, it is crucial that the dual-specification system *helps users without schema knowledge, and potentially even without any SQL experience, correctly construct their desired query.* Our aim is to develop a system that can help non-technical users with domain knowledge to construct expressive SQL queries without the need to consult technical experts. In addition, for technical users, such a system can be a useful alternative to manually writing SQL, which often requires the need to manually inspect the database schema.

[1] ⋈: join, σ: selection, γ: grouping/aggregation
[2] NS: no schema knowledge, PT: partial tuples, OW: open-world assumption
[3] SQuID does not support projected aggregates (i.e. in the SELECT clause).

Second, we want to *minimize user effort in using the system.* Dual-specification interaction should help users more efficiently synthesize queries, especially in contrast to existing single-specification approaches such as NLIs or PBE systems.

Finally, we also want *to have our system run efficiently.* This will both enable us to maximize the likelihood of finding the user's desired query within a limited time budget, and minimize the amount of time the user spends idly waiting for the system to search for queries.

Contributions — We offer the following contributions, extending a preliminary version of this work [3]:

(1) We propose the *dual-specification query synthesis* interaction model and introduce the *table sketch query* (TSQ) to enable users with domain knowledge to construct expressive SQL queries more accurately and efficiently than with single-specification approaches.
(2) We present the *guided partial query enumeration* (GPQE) algorithm, which leverages a neural guidance model to enumerate the query search space and *ascending-cost cascading verification* in order to efficiently prune the search space. We implemented DUOQUEST, a *novel prototype dual-specification system* leveraging GPQE.
(3) We present user studies in which dual-specification enables a 62.5% absolute increase in accuracy over NLI and comparable accuracy to a PBE system on a more limited workload for PBE. We also present a simulation study demonstrating a >2x increase in the top-1 accuracy of DUOQUEST over both NLI and PBE.

Organization — In Section 2, we provide an overview of our problem. We then describe our solution approach (Section 3) and system implementation (Section 4). We present our experimental evaluation, including user studies and simulated experiments (Section 5), explore related work (Section 6), discuss limitations of our approach and opportunities for future work (Section 7), and conclude (Section 8).

2 PROBLEM OVERVIEW

2.1 Motivating Example

Consider the following motivating example:

Example 2.1. Kevin wants to query a relational database containing movie information but has little knowledge of SQL or the schema. He issues the following NLQ to a NLI.

NLQ: Show names of movies starring actors from before 1995, and those after 2000, with corresponding actor names, and years, from earliest to most recent.

Sample Candidate SQL Queries:

CQ1: *Meaning*: The names and years of movies released before 1995 or after 2000 starring male actors, with corresponding actor names, ordered from oldest to newest movie.

```
SELECT m.name, a.name, m.year
FROM actor a JOIN starring s ON a.aid = s.aid
  JOIN movies m ON s.mid = m.mid
WHERE a.gender = `male' AND
  (m.year < 1995 OR m.year > 2000)
ORDER BY m.year ASC
```

CQ2: *Meaning*: The names of movies starring actors/actresses born before 1995 or after 2000 and corresponding actor names and birth years, ordered from oldest to youngest actor/actress.

```
SELECT m.name, a.name, a.birth_yr
FROM actor a JOIN starring s ON a.aid = s.aid
  JOIN movies m ON s.mid = m.mid
WHERE a.birth_yr < 1995 OR a.birth_yr > 2000
ORDER BY a.birth_yr ASC
```

CQ3: *Meaning*: The names and years of movies either (a) released before 1995 and starring male actors, or (b) released after 2000; with corresponding actor names, from oldest to newest movie.

```
SELECT m.name, a.name, m.year
FROM actor a JOIN starring s ON a.aid = s.aid
  JOIN movies m ON s.mid = m.mid
WHERE (a.gender = `male' AND m.year < 1995)
  OR m.year > 2000
ORDER BY m.year ASC
```

The NLI returns over 30 candidate queries. CQ3 is his desired query, but it is the 15th ranked query returned by the NLI and not immediately visible in the interface.

Even for a human SQL expert, the NLQ in Example 2.1 is challenging to decipher, as each of the interpretations cannot be ruled out definitively without an explicit means of clarification by the user. In many cases, NLIs may not return the desired query in the top-k displayed results, and users have no recourse other than to attempt to rephrase the NLQ without additional guidance from the system. In addition, leveraging a previous PBE system for Example 2.1 would be difficult unless Kevin already has a large number of exact, complete example tuples on hand.

With access to DUOQUEST, our dual-specification interface, Kevin can supply an optional PBE-like specification called a *table sketch query (TSQ)* to clarify his query, even with limited example knowledge:

Example 2.2. Kevin chooses to *refine* his natural language query with a table sketch query (TSQ) on DUOQUEST.

He thinks of movies he knows well, and recalls that Tom Hanks starred in Forrest Gump before 1995 and that Sandra Bullock starred in Gravity sometime between 2010 and 2017. He encodes this information in the TSQ shown in Table 2.

Using the NLQ along with the TSQ, the system can eliminate CQ1 because it does not produce the second tuple (with Sandra Bullock, a female, starring in the movie), as well as CQ2, because Sandra Bullock was not born between 2010 and 2017. CQ3 is therefore correctly returned to Kevin.

Types	text	text	number
Tuples			
1.	Forrest Gump	Tom Hanks	
2.	Gravity	Sandra Bullock	[2010,2017]
Sorted?	✗		
Limit?	None		

Table 2: Example table sketch query (TSQ). Top: contains the data types for each column; Middle: example tuples; Bottom: indicates that desired query output will neither be sorted nor limited to top-k tuples.

The TSQ requires *no schema knowledge* from the user, allows users to specify *partial tuples*, and permits an *open-world setting*. When used alone, the TSQ is still likely to face the problem of an intractably large search space. However, when used together with an NLQ, the information from the natural language can guide the process to enable the synthesis of more *expressive queries* such as those including grouping and aggregates.

While the TSQ is optional, a dual-specification input is also preferred over the NLQ alone because it enables *pruning* of the search space of partial queries and permits a *soundness guarantee* that all returned results must satisfy the TSQ. In addition, the TSQ enables users a reliable, alternative means to *refine queries iteratively* (by adding additional tuples and other information to the TSQ) if their initial NLQ fails to return their desired query.

2.2 Table Sketch Query

We formally define the *table sketch query (TSQ)*, which enables users to specify constraints on their desired SQL query at varied levels of knowledge in a similar fashion to existing PBE approaches [13, 14]. Unlike existing approaches, we also allow the user to include some additional metadata about their desired SQL query:

Definition 2.3. A **table sketch query** $\mathcal{T} = (\alpha, \chi, \tau, k)$ has:

(1) an *optional list of type annotations* $\alpha = (\alpha_1, \ldots, \alpha_n)$;
(2) an *optional list of example tuples* $\chi = (\chi_1, \ldots, \chi_n)$;
(3) a *boolean sorting flag* $\tau \in \{\top, \bot\}$ indicating whether the query should have ordered results; and
(4) an *limit integer* $k \geq 0$ indicating whether the query should be limited to the top-k rows[4].

A tuple in the result set of a query, $\chi_q \in R(q)$, **satisfies** an example tuple χ_i if each cell $\chi_q[j] \in \chi_q$ matches the

[4] $k = 0$ indicates no limit.

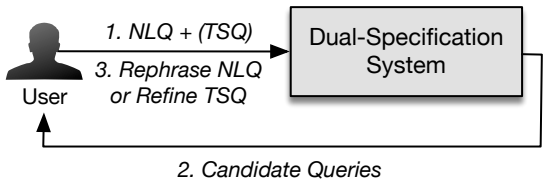

Figure 1: Dual-specification interaction model.

corresponding cell of the same index $\chi_i[j] \in \chi_i$. As shown in Example 2.2, each example tuple $\chi_i \in \chi$ may contain *exact* cells, which match cells in χ_q of the same value; *empty* cells, which match cells in χ_q of any value, and *range* cells, which match cells in χ_q that have values within the specified range.

Definition 2.4. A query q **satisfies** a TSQ $\mathcal{T} = (\alpha, \chi, \tau, k)$ if all of the following conditions are met:

(1) if $\alpha \neq \emptyset$, the projected columns of q must have data types matching the annotations;
(2) if $\chi \neq \emptyset$, for each example tuple in χ, there exists a *distinct* tuple in the result set of q that satisfies it;
(3) if $\tau = \top$, q must include a sorting operator and produce the satisfying tuples in (2) in the same order as the example tuples in the TSQ;
(4) if $k > 0$, q must return at most k tuples.

We denote a table sketch query $\mathcal{T}(q, D)$ as a function taking a query q and database D as input. This function returns $\top$ if executing q on D satisfies $\mathcal{T}$, and $\bot$ otherwise.

2.3 Problem Definition

We now formally define our dual-specification problem:

PROBLEM. *Find the desired query $\hat{q}$ on database D, given:*

(1) a natural language query N describing $\hat{q}$, which includes a set of text and numeric literal values L used in $\hat{q}$;
(2) an optional table sketch query $\mathcal{T}$ such that $\mathcal{T}(\hat{q}, D) = \top$.

The literal values L are a subset of tokens in the natural language query N. These can be obtained from the user by presenting an autocomplete-based tagging interface, as described further in Section 4.

2.4 Interaction

Figure 1 depicts the interaction model. The user issues a NLQ to the system, along with an optional TSQ. The system returns a ranked list of candidate queries. If none of candidate queries is the user's desired query, the user has two options: they may either *rephrase* their NLQ or *refine* their query by adding more information to the TSQ. This process continues iteratively until the user obtains their desired query.

2.5 Task Scope

We consider select-project-join-aggregate (SPJA) queries, including grouping, sorting, and limit operators. In clauses with multiple selection predicates, we disallow nested expressions with different logical operators such as `a > 1 OR (b < 1 AND c = 1)` due to the challenge of expressing such predicates in a NLQ. For simplicity, we restrict join operations to inner joins on foreign key-primary key relationships, although alternate joins such as left joins can also be considered with minimal engineering effort.

3 SOLUTION APPROACH

3.1 Overview

The search space of possible SQL queries in our setting is enormous[5], with a long chain of inference decisions to be made about the presence of clauses, number of database elements in each clause, constants in expressions, join paths, etc. Discovering whether a single satisfying query exists for a set of examples, even in the context of select-project-join queries, is NP-hard [21]. The set of queries we hope to support only further expands this search space.

Previous work [20] attempts to tackle this challenge by implementing beam search, which limits the set of possible generated candidate queries to the k highest-confidence branches at each inference step. However, this approach sacrifices completeness and can cause the correct query to be eliminated in cases where the model performs poorly.

By including the TSQ as an additional specification, we have an alternative means to prune the search space without sacrificing completeness. Consequently, we propose *guided partial query enumeration (GPQE)*, which has two major features. First, GPQE performs *guided enumeration* by using the NLQ to guide the candidate SQL enumeration process, where candidates more semantically relevant to the NLQ are enumerated first. Second, GPQE leverages *partial queries (PQs)* as opposed to complete SQL queries to facilitate efficient pruning, defined as follows:

Definition 3.1. A **partial query (PQ)** is a SQL query in which a query element (i.e. SQL query, clause, expression, column reference, aggregate function, column reference, or constant) may be replaced by a placeholder.

Many NLI systems already generate PQs during query inference [22] or can be easily adapted [20] to do so. These PQs are tested against the TSQ to prune large branches of invalid queries early without needing to enumerate all complete queries in each branch, which is costly both because of the volume of complete queries and the time needed to verify

[5] $O(c^n)$, where $c \geq 2$ is a constant determined by permitted expressivity and n is the number of columns in the schema.

Algorithm 1 Guided Partial Query Enumeration

```
1:  function EnumerateNext(N, M, T, D)
2:      P ← {(∅, 1)}
3:      while P ≠ ∅ do
4:          p ← pop highest priority element from P
5:          Q ← EnumNextStep(p, N, M, D)
6:          for q ∈ Q do
7:              if Verify(T, q[0], D) = ⊥ then
8:                  continue
9:              else
10:                 if q[0] is complete then
11:                     emit q[0] as a candidate query
12:                 else
13:                     push q onto P
```

each one. Ultimately, this enables the approach to cover more of the search space in a given amount of time.

3.2 Algorithm

Algorithm 1 describes the SPQE process, which takes in the natural language query N, an enumeration guidance model M, the table sketch query $\mathcal{T}$, and the database D. P stores the collection of states to explore, where each state is a pair comprised of a partial query and a confidence score for that partial query (Line 2). On each iteration, p, the highest confidence state from P is removed (Line 4). EnumNextStep produces Q, the set of new partial query/confidence score states that can be generated by making an incremental update to a single placeholder on the partial query in p (Line 5). Each state $q \in Q$ is then verified against the table sketch query $\mathcal{T}$ (Line 7), and those that fail verification are discarded. The remaining states are examined to see whether they are complete queries (Line 10), in which case they are emitted as a valid candidate query (Line 11). Otherwise, they are pushed back onto P for another iteration (Line 13). The candidate queries are returned to the user as a ranked list ordered from highest to lowest confidence score.

Figure 2 displays an example GPQE execution, where each box represents a state. Each new layer is an iteration, where candidate states are generated by EnumNextStep using the highest-confidence state available at that iteration. Shaded boxes indicate that the state failed Verify. The highest-ranked candidate query is bolded.

3.3 Guided Enumeration

In this section, we describe the enumeration process in EnumNextStep. We adopt the SyntaxSQLNet [23] system and make several modifications to enable our approach to: (1) perform a complete enumeration over the possible search space, (2) perform a best-first search and robustly compare any two

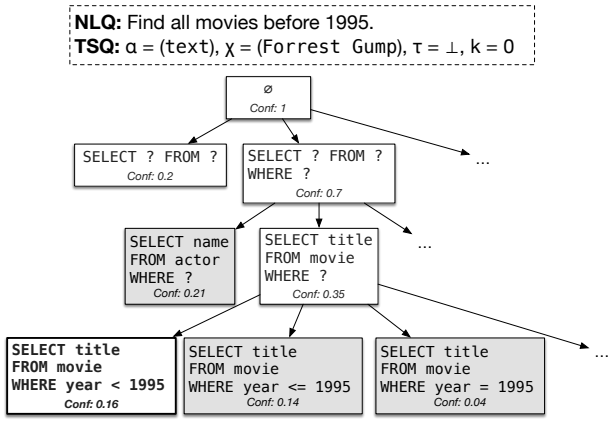

Figure 2: Simplified GPQE example. Each box is a state. Shaded boxes fail verification against the TSQ. The bolded state is the highest-ranked candidate query.

search states during enumeration, (3) perform verification of partial queries by fleshing out their join paths.

We begin by providing some necessary background knowledge of the SyntaxSQLNet system.

3.3.1 Background. SyntaxSQLNet uses a collection of recursive neural network modules, each responsible for making an enumeration decision for a specific SQL syntax element. Each module takes the natural language query N, the partial query synthesized so far p, and optionally, the database schema D. Given the input, each module returns the highest-confidence output class. For modules returning a set as output, a three-step decision is made: (1) a classifier predicts the number of values k to return, (2) another classifier ranks the relevant output classes, and (3) the top-k ranked classes are returned by the module.

3.3.2 Candidate Enumeration. SyntaxSQLNet, by design, produces a single output query as output. To enable the search space enumeration in EnumNextStep to be complete, we modify the modules in SyntaxSQLNet to produce all possible candidate states. We accomplish this by generating a new state for each candidate during each inference decision. For example, when executing the AND/OR module, we generate two candidate states, one each for AND and OR. For modules returning a set as output, the set of returned candidate states is the power set of the output classes.

3.3.3 Confidence Scores. SyntaxSQLNet produces rankings for each state with respect to its siblings in the search space by using the *softmax* function to produce a score in $(0, 1)$ for each output class. However, to facilitate the best-first search in Line 4 of Algorithm 1, we need a overall confidence score that enables us to compare two states even if they are not siblings. As a result, we explicitly define the confidence score

C for a partial query state p as follows:

$$C(p) = \prod_{i=1}^{|p|} M(N, p_i, D)$$

where each p_i is the output class of the i-th inference decision made to generate the partial query in state p, and $M(N, p_i, D)$ is the softmax value returned by the appropriate SyntaxSQLNet module for NLQ N, output class p_i, on the schema of database D. In other words, the confidence score is the cumulative product of the *softmax* values of each output class comprising the partial query. Defining the confidence score in this way guarantees the following property:

Property 1. *The sum of the confidence scores of all child branches of state p is equal to the confidence score of p.*

In theory, this confidence score definition also causes the system to prefer shorter queries over longer ones. Such concerns motivate previous systems [22] to adopt a confidence score definition motivated by the geometric mean. In practice, however, we found that this property of our confidence score did not negatively affect our system's ability to accurately synthesize user queries.

3.3.4 Progressive Join Path Construction. SyntaxSQLNet includes a rudimentary join path inference module to determine the tables and join conditions used in the `FROM` clause of a query. In SyntaxSQLNet, this join path module is (1) only applied to completed queries as the final step in the query inference process, and (2) only produces a single join path.

For our GPQE algorithm, however, we need join paths to be produced for each partial query, because the Verify procedure needs to be able to execute partial queries to compare them against the example tuples in the TSQ. In addition, user-provided NLQs often lack explicit information to guide the system to select one particular join path over another [2]. For this reason, and also to enable completeness in our search procedure, we produce all candidate join paths for each partial query rather than just a single join path.

To accomplish these goals, we adopt *progressive join path construction*. Algorithm details are given in [4].

3.3.5 Extensibility. As NLI models are undergoing rapid active development in the programming languages [22], natural language processing [5, 7, 23], and database research communities [8], our approach is modular, enabling SyntaxSQLNet to be replaced by any NLI model that:

(1) is able to generate and incrementally apply updates to executable partial queries,
(2) emits a confidence score for each partial query in the range $[0, 1]$ and fulfilling Property 1.

3.3.6 Scope. While SyntaxSQLNet supports set operations (`INTERSECT`, `UNION`, `EXCEPT`) and nested subqueries in predicates, we disabled this functionality to restrict output to the tasks described in Section 2.5.

3.4 Verification

During the enumeration process, verifying queries against the TSQ can be expensive for two reasons: (1) waiting until candidate queries are completely synthesized before verification causes redundant work to be performed on similar candidate queries, and (2) executing a single, complete candidate query on the database can be costly depending on the nature of the query and the database contents.

To mitigate these inefficiencies, we leverage *ascending-cost cascading verification* for the Verify function in Algorithm 1. Low-cost verifications, which do not require access to the database D, are performed first to avoid performing high-cost verifications, which involve issuing queries on D, until necessary. In addition, these verifications are performed as early as possible on partial queries in order to avoid performing redundant work on similar candidate queries. Full details and a running example are provided in [4].

3.5 Alternative Approaches

Two naïve approaches to designing a dual-specification system are (1) *intersecting* the output of an NLI and PBE system and (2) *chaining* two systems so the output of one becomes the input of the next. The intersection approach is inefficient because each system will have to redundantly examine the search space without communicating with the other system. The chaining approach is more promising, where candidate queries generated by a NLI can be passed to a PBE system for verification, eliminating the redundancy in the intersection approach. However, it is still inefficient in comparison to GPQE, which enables us to eliminate large branches of complete queries by pruning partial queries.

4 IMPLEMENTATION

We implemented our approach in a prototype system, Duoquest[6], with user interface shown in Figure 3. Notably, users can trigger autocomplete functionality over a master inverted column index [16] for domain-specific literal text values both in the NLQ search bar and in each cell of the TSQ table. Further details are provided in [4].

After issuing a query, candidate SQL queries are displayed one at a time from highest to lowest confidence as the system enumerates and verifies them. SQL queries continue to load until a timeout threshold is exceeded or the user clicks "Stop Task". To assist users in distinguishing candidate queries, each candidate query has a "Query Preview" button which

[6]https://github.com/umich-dbgroup/duoquest

Experiment	Dataset	Databases	Tasks				Avg. Schema Statistics		
			Easy	Med	Hard	Total	Tables	Columns	FK-PK
User Study vs. NLI	MAS [8]	1	0	3	5	8	15	44	19
User Study vs. PBE	MAS [8]	1	0	4	2	6	15	44	19
Simulation	Spider Dev [24]	20	239	252	98	589	4.1	22.1	3.2
	Spider Test [24]	40	524	481	242	1247	4.5	19.6	3.6

Table 3: Datasets used in our experiments, with the number of distinct databases and tasks per dataset, and the average number of tables, columns, and foreign key-primary key (FK-PK) relationships in all schemas. *Easy* tasks were project-join queries including aggregates, sorting, and limit operators, *Medium* tasks also included selection predicates, and *Hard* tasks included grouping operators.

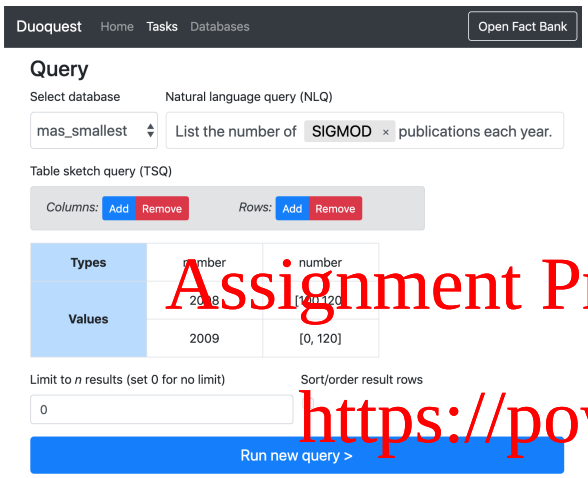

Figure 3: Screenshot of front-end interface. The "SIGMOD" tag was produced via autocomplete.

retrieves a 20-row preview of the query results, and a "Full Query View" which executes the full query on the database.

5 EVALUATION

We explored several research questions in our evaluation, and full results are published in [4]. In this paper, we summarize our main results to two questions:

RQ1: Does the dual-specification approach help users to correctly synthesize their desired SQL query compared to single-specification approaches?

RQ2: Does the dual-specification approach conserve user effort over single-specification approaches?

5.1 Setup for User Studies

5.1.1 Compared Systems. For **RQ1**/**RQ2**, we conducted two within-subject user studies: one between DUOQUEST and SyntaxSQLNet [23], a state-of-the-art NLI; and the other with DUOQUEST and SQuID [6], a state-of-the-art PBE system.

We selected SyntaxSQLNet as a representative end-to-end neural network NLI. While some recent NLIs [5, 7] are known to outperform SyntaxSQLNet, their code was not available at the time of our study. In addition, their contributions are orthogonal to ours and can provide corresponding improvements to the guided enumeration process in DUOQUEST.

We selected SQuID as the representative PBE system because, to the best of our knowledge (Table 1), it is the only prominent PBE system that makes an open-world assumption, does not require schema knowledge of the user, and permits query expressivity beyond projections and joins.

For convenience, we denote SyntaxSQLNet as *NLI* and SQuID as *PBE* for the remainder of this section.

5.1.2 Users. To reflect our motivation of supporting users with no specific knowledge of the schema and potentially without SQL experience, we recruited 16 users with no prior knowledge of the schema for our studies. Six of the users had little to no experience with SQL, while the remaining 10 had at least some experience with SQL.

5.1.3 Tasks. We tested DUOQUEST against NLI on a variety of tasks within the scope described in Section 2.5. Since PBE did not support projected numeric columns or aggregates, we generated a second task set with a more limited scope of tasks for our study comparing DUOQUEST and PBE.

We tested each user on the Microsoft Academic Search (MAS) database[7] (Table 3) to see if they could synthesize the desired SQL query matching the provided task description. Each task description was provided in Chinese[8] This resulted in a total of 128 task trials for the NLI study (64 on each system), and 96 task trials (48 on each system) for the PBE study. Users were given a time limit of 5 minutes for each task trial, which, in practice, was ample time for virtually all users to either complete the trial or give up after losing patience. Each user was given the same 2 tutorial tasks related to the actual task workload to try on each system prior to the study to learn how to use each system.

[7]We removed some rows and columns unused in our tasks from the original database to reduce the user study time.

[8]All recruited subjects were bilingual in Chinese and English. following the study procedure in [8] to force the user to articulate the NLQ in English using their own words.

The tasks (full list in [4]) were split into two sets per user study. Half of the users were each given the first set to perform on DUOQUEST first, then the second set to perform on the baseline system, while the other half of the users first attempted the first set on the baseline system, then the second set on DUOQUEST. The tasks in each set were given in the same order for each system, along with the 2 initial tutorial tasks, so that if there were any learning effects, they would happen equally on both systems. This means that results are comparable across systems for a given task, but not necessarily between two tasks.

5.1.4 Query Selection. NLI and DUOQUEST produced a list of candidate SQL queries ranked from highest to lowest confidence, where each candidate query appeared as soon as the system enumerated it. Users with at least some SQL experience attempted to directly read the SQL queries before selecting one, as they could often understand the semantics of candidate queries even with no prior knowledge of the schema. On the other hand, users with little to no knowledge of SQL selected queries using a combination of eyeballing the selection predicates in the SQL queries and observing the "Query Preview" (described in Section 4) to view a sample of the result set of each candidate query as a sanity check.

In contrast to the other systems, PBE offered an "explanation" interface where users could check/uncheck suggested "filters" (i.e. selection predicates) to modify the produced query, with no need to consider the underlying SQL.

As a result, in the NLI study, both systems equally suffered from the same risk of users failing to properly understand the candidate SQL queries displayed to them. In the PBE study, the explanation interface arguably offered a slight advantage to PBE over DUOQUEST for users with little knowledge of SQL. However, the study results demonstrated that the current interface was sufficient even for users without SQL knowledge to select the correct query on DUOQUEST.

5.1.5 Fact Bank. We designed our studies to explore the usability of each system given a *fixed level of pre-existing domain knowledge* in an open-world setting—i.e. where users only know a proper subset of tuples that will be produced by their desired query. To emulate such domain knowledge, we provided each user with a fact bank of 10 facts per task which was presented in randomly shuffled order during each trial. We allowed them to use any subset of these facts, but we did not allow them to use any knowledge external to the fact bank. These facts could be used in two ways: first, as example tuple input for DUOQUEST or PBE; and second, as a means to verify the results of candidate queries by observing whether the facts reside in the produced output preview.

Each fact was provided as a sentence rather than as a tuple to require the user to discern how to input the fact into each system. For example, "List authors and their number of publications," a fact would be written in the form "Author X wrote 50 to 100 publications," and the user would figure out how to input this as `(X, [50, 100])` into DUOQUEST.

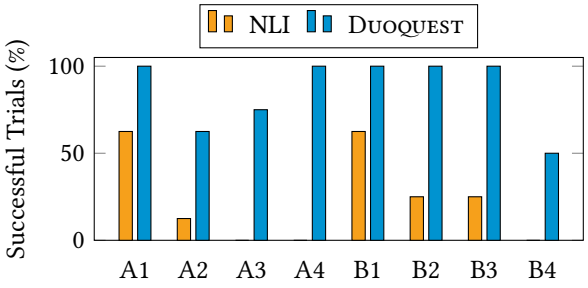

Figure 4: % of trials for NLI study in which the user successfully completed each task within 5 minutes.

A caveat of the fact bank design is that it does not test what happens when users provide incorrect examples. This presents a risk of bias in our NLI study because DUOQUEST leverages facts while NLI does not. On the other hand, in the PBE study, both systems equally benefit from the fact bank. In the real world, the challenge of incomplete user knowledge is mitigated in DUOQUEST by the autocomplete interface and the ability to provide partial or range examples. However, further study is required to better investigate the effects of noisy examples on our system.

5.2 User Study vs. NLI

Figure 4 displays the proportion of the time users successfully completed each task. With regard to **RQ1**, it is clear that DUOQUEST enables users to discover the correct query far more frequently than the baseline NLI system, as only 15 out of 64 (23.4%) trials were successful with NLI while that number shot up to 55 (85.9%) for DUOQUEST, a ***62.5% absolute increase in the percentage of task trials completed correctly***. As evident from the figure, DUOQUEST outperformed NLI on each individual task, with users failing to complete even a single trial on NLI for tasks A3, A4, B4. This is largely due to the additional PBE specification, which drastically shrinks the list of displayed candidate queries for DUOQUEST, while users grow fatigued manually verifying candidate queries in the large list for NLI.

For **RQ2**, we use user time as a metric for user effort, and observe in Figure 5 that ***DUOQUEST either reduces or requires comparable user effort to the baseline NLI system for every successful trial***. This is also due to the reduction in the number of candidate queries displayed to the user.

Finally, the mean number of examples provided to DUOQUEST fell between 1 and 1.5 for each task, suggesting that ***DUOQUEST can be an effective tool for users even with just one or two examples*** regarding their desired query.

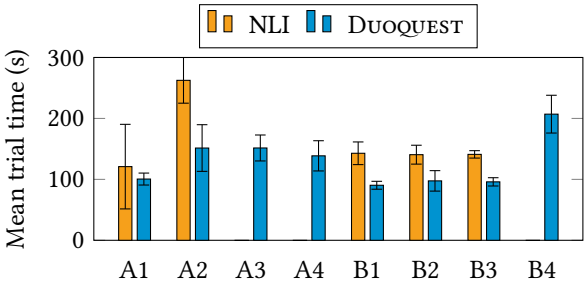

Figure 5: Mean time per task for correctly completed trials in NLI study, with error bars indicating standard error. A3, A4, B4 had no successful trials on NLI.

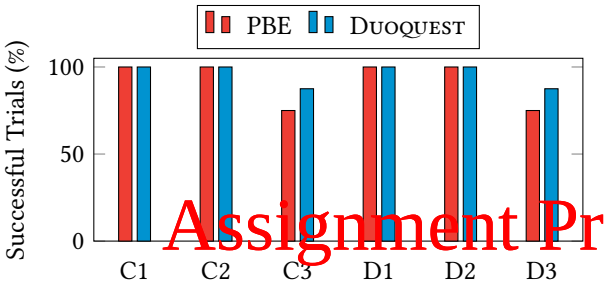

Figure 6: % of trials for PBE study in which the user successfully completed each task within 5 minutes.

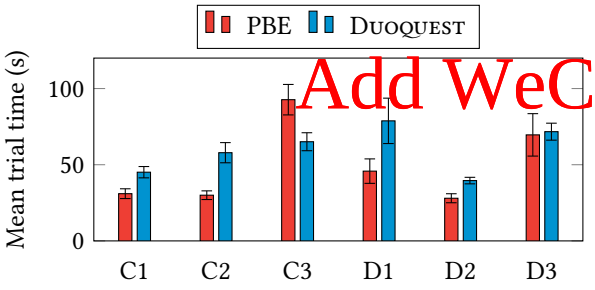

Figure 7: Mean time per task for correctly completed trials in PBE study; error bars for standard error.

5.3 User Study vs. PBE

For **RQ1**, Figure 6 shows ***DUOQUEST and PBE perform comparably on the PBE-supported workload***, with DUOQUEST performing slightly better on Hard tasks (C3, D3).

For **RQ2**, Figure 7 shows that ***user time is comparable for PBE and DUOQUEST on harder tasks but PBE is faster for simple tasks***. PBE was faster for users on the easier Medium-level tasks (C1, C2, D1, D2) because of the time required for users to type out the NLQ on DUOQUEST. This additional cost was amortized for the more difficult Hard tasks (C3, D3) which contained aggregate operations due to the benefits gained by the additional NLQ specification.

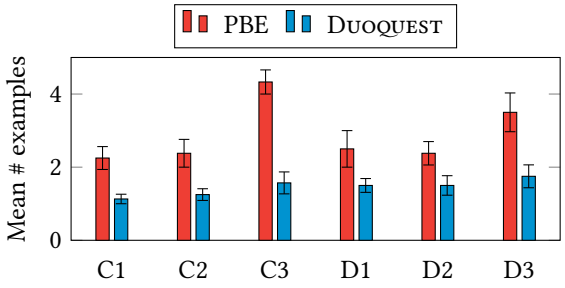

Figure 8: Mean # examples used per task for successful trials in PBE study; error bars for standard error.

Figure 8 displays how users issue more examples on average for PBE, suggesting that ***DUOQUEST may be preferred in cases when users know fewer examples*** if they are able to articulate an NLQ instead.

5.4 Simulation Study

5.4.1 Setup. We evaluated DUOQUEST on the Spider benchmark [24], which is comprised of 10,181 NLQ-SQL pairs on 200 databases split into training (7,000 tasks), development (1,034 tasks), and test (2,147 tasks) sets. We removed tasks for which the SQL produced an empty result set or was outside our task scope (Section 2.5), or if the database had annotation errors (e.g. incorrect data types or integrity constraints in the schema). The final development and test sets we tested on (Table 3) had 589 tasks and 1,247 tasks, respectively.

For each task, the SQL label from the Spider benchmark was designated as the user's desired query, and literal values used within the SQL label were set to be the input literals L. We synthesized TSQs for each task, where each of the TSQs contained type annotations, two example tuples randomly selected from the result set of the desired SQL query, and τ and k values corresponding to the desired query.

We compared the 3 systems from the user studies: DUOQUEST; SyntaxSQLNet (*NLI*); and SQuID (*PBE*). For each task, DUOQUEST was given the NLQ, literals, and synthesized TSQ; NLI was given the NLQ and literals; and PBE was given the example tuples of the synthesized TSQ. The systems were run on the same machines as the user study.

DUOQUEST and NLI produced a ranked list of candidate queries one at a time from highest to lowest confidence. The task was terminated when the desired query was produced by the system or a timeout of 60 seconds was reached. On the other hand, PBE returned a single set of projected columns with multiple candidate selection predicates at a single point in time, with a mean runtime of 1.7 seconds for the development set and 0.7 seconds for the test set.

5.4.2 Accuracy. Reinforcing our conclusions on **RQ1** from the user study, DUOQUEST handily beat NLI and PBE, ***with***

a >2x increase in Top-1 accuracy and 47.6% increase in Top-10 accuracy over NLI, and a larger improvement over PBE on the development set. Results were similar on the test set. Detailed results are in [4].

The PBE system was unable to handle a large proportion of our benchmark tasks because it did not support projections of numeric columns or aggregate values nor selection predicates with negation or LIKE operators. While PBE should have been able to get the remaining supported tasks correct, it failed several tasks due to its requirements for a snowflake schema and user-defined metadata annotations as to which schema attributes are "entities" or "concepts". While we offered our best effort in restructuring the schema to support all given tasks, we found that for some schemas, all tasks for the schema could not be simultaneously supported with any schema structure given the current system design.

6 RELATED WORK

Natural language interfaces. Most early natural language interfaces for relational databases were confined to a single domain [1]. Later work focused on the general-purpose case for easy adoption on arbitrary schemas. The Precise system explicitly defined "semantic coverage" to constrain the scope of natural language that could be expressed [17]. Other systems utilized different technologies such as dependency parse trees [8], semantic parsing [22], or pre-defined ontologies [15] to expand the scope of expressible queries. More recently, advances in deep learning have given rise to a new approach of building end-to-end deep learning systems to translate natural language queries to SQL. The current state-of-the-art utilizes techniques such as a modular syntax tree network [23], graph neural networks [5], or an intermediate representation [7] to generate SQL queries of arbitrary complexity. Our dual-specification approach alleviates ambiguity in natural language by allowing the user to provide a table sketch query to constrain the query search space.

Programming-by-example (PBE) systems. These interfaces permit users to provide a set of example output tuples or the full output of the desired query to search for queries on the database. A large body of work exists in this area [10], a representative sample of which is displayed in Table 1. Such systems often have to sacrifice query complexity or enforce requirements on user knowledge (schema knowledge; full, exact tuples; or a closed-world setting) to make the search problem tractable. More recent work [6] has made an attempt to discern query intent in PBE with complex queries using pre-computed statistics and semantic properties. Our dual-specification approach tackles the same challenge in an orthogonal manner by leveraging the user's natural language query in addition to the user-provided examples.

7 LIMITATIONS AND FUTURE WORK

In this section, we identify some potential limitations and improvements to the current DUOQUEST prototype.

First, additional work needs to be done to produce a *completely SQL-less interaction model*. Currently, users interact with produced candidate SQL queries to select their final query. During our evaluation, users without knowledge of SQL or the schema used various signals to assess whether a candidate query was the desired one (Section 5.1.4), and they were for the most part successful. Users' success may vary, however, when working with schemas with confusing attribute names or with highly complex SQL queries. As a result, there is a need for an interaction model that permits users to validate candidate SQL queries with their domain knowledge without exposing the actual SQL syntax to them.

Second, DUOQUEST is not yet able to deal with *noisy (i.e. incorrect) examples*. In the real world, users are often prone to errors and misinformation, and while this is mitigated somewhat by the autocomplete feature in DUOQUEST, techniques such as error detection or probabilistic reasoning should be implemented to enable DUOQUEST to handle noisy examples.

Finally, DUOQUEST can be improved by *streamlining iterative interaction*. For example, the current interface could be improved by enabling users to add positive or negative examples to the TSQ specification by clicking a button directly on a candidate query preview. In addition, enabling users to directly modify generated candidate queries, perhaps by presenting them in some intermediate representation, would allow greater flexibility in synthesizing queries than merely having the user select from the system-generated list.

8 CONCLUSION

In this paper, we proposed dual-specification query synthesis, which consumes both a NLQ and an optional PBE-like table sketch query enabling users to express varied levels of knowledge. We introduced the guided partial query enumeration (GPQE) algorithm to synthesize queries from a dual-mode specification, and implemented GPQE in a novel prototype system DUOQUEST. We presented results from a user study in which DUOQUEST enabled a 62.5% absolute increase in query construction accuracy over a state-of-the-art NLI and comparable accuracy to a PBE system on a more limited workload supported by the PBE system. In a simulation study, DUOQUEST demonstrated a >2x increase in top-1 accuracy over both NLI and PBE.

ACKNOWLEDGEMENTS

We are grateful for a University of Michigan MIDAS grant to fund this work. We also thank Tao Yu, Bo Pang, and the Yale LILY Lab for assisting us with executing DUOQUEST on the Spider dataset.

REFERENCES

[1] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch. Natural language interfaces to databases - an introduction. *Natural Language Engineering*, 1(1):29–81, 1995.

[2] C. Baik, H. V. Jagadish, and Y. Li. Bridging the semantic gap with SQL query logs in natural language interfaces to databases. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 374–385. IEEE, 2019.

[3] C. Baik, Z. Jin, M. J. Cafarella, and H. V. Jagadish. Constructing expressive relational queries with dual-specification synthesis. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*, 2020.

[4] C. Baik, Z. Jin, M. J. Cafarella, and H. V. Jagadish. Duoquest: A dual-specification system for expressive SQL queries. *CoRR*, abs/2003.07438, 2020.

[5] B. Bogin, J. Berant, and M. Gardner. Representing schema structure with graph neural networks for text-to-SQL parsing. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 4560–4565, 2019.

[6] A. Fariha and A. Meliou. Example-driven query intent discovery: Abductive reasoning using semantic similarity. *PVLDB*, 12(11):1262–1275, 2019.

[7] J. Guo, Z. Zhan, Y. Gao, Y. Xiao, J. Lou, T. Liu, and D. Zhang. Towards complex text-to-sql in cross-domain database with intermediate representation. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 4524–4535, 2019.

[8] F. Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *PVLDB*, 8(1):73–84, 2014.

[9] H. Li, C. Chan, and D. Maier. Query from examples: An iterative, data-driven approach to query construction. *PVLDB*, 8(13):2158–2169, 2015.

[10] D. M. L. Martins. Reverse engineering database queries from examples: State-of-the-art, challenges, and research opportunities. *Inf. Syst.*, 83:89–100, 2019.

[11] K. Panev and S. Michel. Reverse engineering top-k database queries with PALEO. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016.*, pages 113–124, 2016.

[12] A. Popescu, O. Etzioni, and H. A. Kautz. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th International Conference on Intelligent User Interfaces, IUI 2003, Miami, FL, USA, January 12-15, 2003*, pages 149–157, 2003.

[13] F. Psallidas, B. Ding, K. Chakrabarti, and S. Chaudhuri. S4: top-k spreadsheet-style search for query discovery. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 2001–2016, 2015.

[14] L. Qian, M. J. Cafarella, and H. V. Jagadish. Sample-driven schema mapping. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 73–84, 2012.

[15] D. Saha, A. Floratou, K. Sankaranarayanan, U. F. Minhas, A. R. Mittal, and F. Özcan. ATHENA: an ontology-driven system for natural language querying over relational data stores. *PVLDB*, 9(12):1209–1220, 2016.

[16] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik. Discovering queries based on example tuples. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 493–504, 2014.

[17] W. C. Tan, M. Zhang, H. Elmeleegy, and D. Srivastava. REGAL+: reverse engineering SPJA queries. *PVLDB*, 11(12):1982–1985, 2018.

[18] Q. T. Tran, C. Y. Chan, and S. Parthasarathy. Query reverse engineering. *VLDB J.*, 23(5):721–746, 2014.

[19] C. Wang, A. Cheung, and R. Bodík. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 452–466, 2017.

[20] C. Wang, P. Huang, A. Polozov, M. Brockschmidt, and R. Singh. Execution-guided neural program decoding. *CoRR*, abs/1807.03100, 2018.

[21] Y. Y. Weiss and S. Cohen. Reverse engineering SPJ-queries from examples. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 151–166, 2017.

[22] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig. SQLizer: query synthesis from natural language. *PACMPL*, 1(OOPSLA):63:1–63:26, 2017.

[23] T. Yu, M. Yasunaga, K. Yang, R. Zhang, D. Wang, Z. Li, and D. R. Radev. 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, Brussels, Belgium, October 31 - November 4, 2018*, pages 1653–1663, 2018.

[24] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. R. Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pages 3911–3921, 2018.

[25] M. M. Zloof. Query by example. In *American Federation of Information Processing Societies: 1975 National Computer Conference, 19-22 May 1975, Anaheim, CA, USA*, pages 431–438, 1975.