Constructing Expressive Relational Queries with Dual-Specification Synthesis

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 the user to have a grasp of the relational model, the SQL language, and the schema at hand. While natural language interfaces (NLIs) and programming-by-example (PBE) are promising alternatives, they suffer from various challenges. Natural language queries (NLQs) are often ambiguous, even for human interpreters, and current PBE approaches require either low-complexity queries, user schema knowledge, exact example tuples from the user, or a closed-world assumption 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 knowledge. We introduce the Duoquest system, which leverages *guided partial query enumeration* to efficiently explore the space of possible queries. We demonstrate in experiments on the prominent Spider benchmark that Duoquest substantially outperforms state-of-the-art NLI and PBE approaches.

1 INTRODUCTION

Querying a relational database is difficult because it requires the user to have a grasp of the relational model, the SQL language, and the schema at hand. Consequently, an ongoing research challenge is to enable users who lack such knowledge 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 [4, 7, 11] or are overly reliant on specific sentence structures [2]. More recent advances leverage deep learning to circumvent these challenges, but the state-of-the-art accuracy [12] on established benchmarks falls well short of the desired outcome, which is that NLIs can only misinterpret users' questions very rarely, if at all [4]. Moreover, this problem is exacerbated by the fact that natural language queries (NLQs) often contain inherent ambiguities [1] such as in the following situation:

Example 1.1. 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 of any gender born before 1995 or after 2000 and corresponding actor names and birth years, ordered from oldest to youngest actor.

```
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
```

Even for a human SQL expert, the NLQ in Example 1.1 is challenging to decipher, as each of the interpretations cannot be ruled out definitively without explicit clarification from 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 [4] without additional guidance from the system.

Another alternative to writing raw 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. Unlike NLIs, these approaches have the advantage of a concrete notion of *soundness* in that returned candidate queries are guaranteed to satisfy the user-provided specification. However, as shown in Table 1, PBE systems must precariously juggle four factors: (1) how much *query complexity* is permitted, (2) wheter *schema knowledge* is required of the user, (3) whether users must provide *full tuples* or can provide partial ones, and (4) whether an *open- or closed-world setting* is assumed, where the closed-world setting entails that the system assumes that the user has provided a complete result set instead of a subset of possible returned tuples.

The ideal PBE system would produce complex queries with aggregates and nesting while enabling users to provide partial tuples in an open-world setting without schema knowledge. However,

	Query Complexity[1]				User Knowledge		
System	π/⋈	σ	γ	Nest	Schema	Tuples	World
NLIs [2, 7, 11, 12]	✓	✓	✓	✓		N/A	N/A
PBE Systems							
QBE [14]	✓	✓	✓	✓	Need	Partial	Open
MWeaver [6]	✓					Full	Open
FastTopK [5]	✓					Partial	Open
TALOS [9]	✓	✓	✓			Full	Closed
QFE [3]	✓	✓			Need	Full	Closed
Scythe [10]	✓	✓	✓	✓	Need	Full	Closed
REGAL+ [8]	✓	✓	✓			Full	Closed
Duoquest	✓	✓	✓	✓		**Partial**	**Open**

Table 1: Duoquest versus existing NLI/PBE systems.

Types	`string`	`string`	`number`
Tuples			
1.	Forrest Gump	Tom Hanks	
2.	Gravity	Sandra Bullock	[2010, 2017]
Sorted?	✗		
Limit?	∞		

Table 2: Example table sketch query (TSQ).

previous systems could not handle the massive search space of this scenario and each constrained at least one of the above factors.

Our Approach — We observe that PBE and NLQ specifications can be leveraged in a synergistic manner, 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, including 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 called a *table sketch query (TSQ)* as input.

Example 1.2. Kevin issues the NLQ in Example 1.1, and the NLI returns several 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. Instead of manually sifting through all the candidate queries, he chooses to *refine* the query with a TSQ.

He thinks of movies he knows well. Specifically, he knows 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. The top section contains the data types for each column, the middle section contains his knowledge in the form of example tuples, and the bottom section indicates that his desired query's output will neither be sorted nor limited to top-k tuples.

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.

[1] π: projection, ⋈: join, σ: selection, γ: grouping/aggregation

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 a 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 aggregates and nesting.

While the TSQ is optional, a dual specification is also preferred over the NLQ alone because it 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. In addition, the TSQ enables ***pruning*** of the search space of partial queries and permits a ***soundness guarantee*** that all returned results must satisfy the TSQ.

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

First, it is not obvious *how to integrate single-specification approaches*. Naïve approaches include computing the intersection of candidate output sets from two single-specification systems, or chaining two systems such that the output of one becomes the input of the other. These approaches, however, miss out on the opportunity for early pruning offered by two specifications.

Second, it is important to *facilitate varying degrees of user domain knowledge*. While it is reasonable to expect that users are able to provide a NLQ for their desired query in every case, the amount of user knowledge that can be specified in a TSQ may vary widely depending on the task and the user's familiarity with the domain. The challenge then is to enable users to specify varying amounts of knowledge, and to maximize the impact of whatever knowledge is given for more accurate query synthesis.

Finally, we aim *to have our system run fast with low memory needs*. This will enable us to maximize the likelihood of finding the user's desired query within a limited time and memory budget.

Contributions — We offer the following contributions:

(1) We propose *dual-specification query synthesis* to enable users with varied domain knowledge to construct expressive SQL queries with a NLQ and TSQ.
(2) We efficiently explore the space of possible queries with *guided partial query enumeration* and implement this in a prototype system, Duoquest.
(3) We present selected experiments on the prominent Spider benchmark in which Duoquest produced the user's desired query within 60 seconds 83.7% of the time, improving on state-of-the-art NLI and PBE approaches which attained 56.7% and 34.0%, respectively.

2 OVERVIEW

2.1 Preliminaries

We first explain some preliminary concepts. All concepts and definitions provided are in the context of a fixed existing database.

We begin by defining the *table sketch query (TSQ)*, which enables users to specify constraints on their desired query at varied levels of knowledge in a similar fashion to existing PBE approaches [5, 6]:

Definition 2.1. A **table sketch query** $\mathcal{T} = (\alpha, \chi, \tau, k)$ has:

(1) an *optional list of type annotations* $\alpha = (\alpha_1, \ldots, \alpha_n)$;

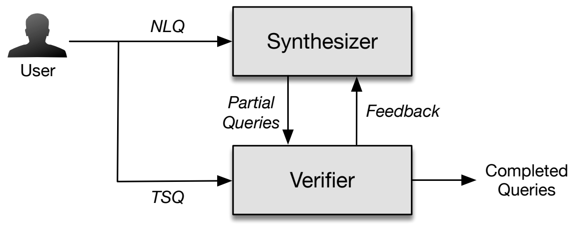


Figure 1: Guided partial query enumeration.

(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[2].

A tuple in the result set of a query, $\chi_q \in R(q)$, **satisfies** χ_i if each cell $\chi_q[j] \in \chi_q$ matches the corresponding cell of the same index $\chi_i[j] \in \chi_i$. As shown in Example 1.2, 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 2.2. 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.

2.2 Problem Definition

We now formally define our dual-specification problem:

Problem. *Find the user's desired query $\hat{q}$, given:*
(1) a natural language query N describing $\hat{q}$;
(2) an optional table sketch query $\mathcal{T}$ such that $\mathcal{T}(\hat{q}) = \top$.

2.3 Interaction

The user begins by issuing 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.

3 APPROACH

3.1 Guided Partial Query Enumeration

The search space of possible SQL queries in our setting is enormous, 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. Given the number of options possible for each inference decision, enumerating and verifying every possible complete SQL query against the TSQ independently can consume a large amount of time and memory.

We propose *guided partial query enumeration (GPQE)* as a solution to this challenge, 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)*—i.e. SQL queries with holes/placeholders for various elements—as opposed to complete queries. Several existing NLIs [11, 12] can easily be modified to enumerate candidate PQs in intermediate stages before the queries are complete. These PQs can be tested against the TSQ to prune large branches of invalid queries early without enumerating all complete queries in each branch. This enables the search to cover a wider proportion of the expansive space of possible queries in a given amount of time.

Figure 1 provides an overview of GPQE. The Synthesizer module guides enumeration by emitting PQs most relevant to the NLQ first. The Verifier module receives the TSQ and each PQ and determines whether the PQ satisfies the TSQ and sends one of three feedback signals to the Synthesizer: $\top$, meaning that any completion of the PQ will satisfy the TSQ; $\bot$, meaning that no completion of the PQ will satisfy the TSQ; or ?, meaning that no conclusion can yet be drawn. The Synthesizer continues enumerating more fleshed out versions of the PQ as long as the feedback is not $\bot$. The Verifier returns any queries evaluated as $\top$.

3.2 Implementation

We implemented GPQE in a prototype system, Duoquest. The Synthesizer in Duoquest uses a pre-trained neural network model from an existing NLI [12]. Given the modular construction of Duoquest, this Synthesizer can easily be replaced with another model or NLI so long as the NLI is able to synthesize syntactically correct PQs in order of most to least confidence.

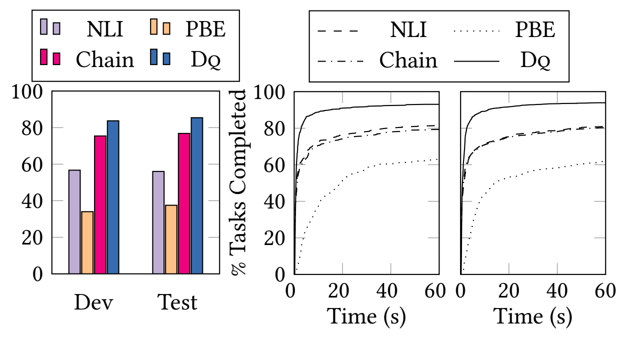
The Verifier in Duoquest performs rule-based pruning which extends the techniques in [10] to an open-world setting enabling partial tuples and requiring no schema knowledge. Specifically, [10] proposed an *over-approximation property* for a partial query $\widetilde{q}$ in which any completion q of $\widetilde{q}$ is guaranteed to have a result set that is a subset of the result set of $\widetilde{q}$. We developed novel pruning rules for our setting that preserve the over-approximation property.

Duoquest supports select-project-join-aggregate (SPJA) queries with `ORDER BY` and `LIMIT` clauses, set operations (union, intersect, except), and nested subqueries in selection clauses. To our knowledge, it is the first system to support this level of query complexity without requiring schema knowledge or a closed-world setting.

3.3 Alternative Approaches

Two naïve approaches to designing a dual-specification system are (1) *intersecting* the output of two single-specification systems 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

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



(a) Top-10 Acc. (%) **(b) Dev** **(c) Test**

Figure 2: Experiments on Spider. In (b) and (c), a higher curve indicates superior performance.

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 verify and eliminate many potential SQL queries as opposed to one at a time.

4 ONGOING EXPERIMENTS

We evaluated Duoquest on Spider [13], which is comprised of 10,181 NLQ-SQL pairs on 200 databases split into training, development, and test sets. We used the training set to train the Synthesizer model and ran our experiments on each of the development and test sets. We removed tasks for which the SQL produced an empty result set or was outside our task scope of unnested select-project-join-aggregate (SPJA) queries with grouping, sorting, and limit operators, and synthesized TSQs for each of the remaining tasks. Each of the synthesized TSQs contained type annotations, two example tuples for tasks with `ORDER BY` clauses and one example tuple otherwise, and τ and k values derived from the SQL. The final development and test sets had 589 and 1,247 tasks, respectively.

We compared four systems:

(1) Duoquest, using both the NLQ and synthesized TSQ;
(2) *NLI*, a state-of-the-art NLI [12] using only the NLQ;
(3) *PBE*, using only the TSQ with breadth-first search enumeration; and
(4) *Chain*, which implements the chaining approach in Section 3.3 and uses both the NLQ and TSQ.

For each task, the SQL label was designated as the user's desired query $\hat{q}$, and we enforced a timeout of 60 seconds. Results are displayed in Figure 2 for the development and test sets; we discuss only the development set as results are similar for both sets.

In Figure 2a, **Duoquest attained 83.7% top-10 accuracy on the development set while NLI, the best-performing single-specification system, only achieved 56.7%**. While Chain would produce an identical result to Duoquest in a scenario with unlimited time, it achieved a lower accuracy of 75.4% due to the 60-second time limit, demonstrating the promise of GPQE to increase the efficiency of the dual-specification approach. PBE performed poorly at 34.0% because the search space of possible SQL syntax was too large for unguided enumeration within the given time limit.

Similar results are conveyed by Figure 2b and 2c, which display the distribution of when each system generated the user's desired query, regardless of ranking, over time. Duoquest's GPQE implementation clearly outperforms Chain from an efficiency perspective. In addition, while the final top-10 accuracy of Chain is significantly higher than that of NLI in Figure 2a, Chain performs slightly worse than NLI in Figure 2b and 2c because NLI is able to explore more candidate queries without the verification overhead of adding an additional specification.

5 CONCLUSION AND FUTURE WORK

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 tackled this problem with guided partial query enumeration (GPQE), and implemented GPQE in Duoquest. We demonstrated in experiments on the Spider benchmark that Duoquest produced the user's desired query within 60 seconds 83.7% of the time, improving on state-of-the-art NLI and PBE approaches which attained 56.7% and 34.0%, respectively. For future work, we will improve the performance of Duoquest, expand our experiments, and perform a user study to evaluate the viability of our interaction model.

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] K. Church and R. Patil. Coping with syntactic ambiguity or how to put the block in the box on the table. *Comput. Linguist.*, 8(3-4):139–149, July 1982.
[2] F. Li and H. Jagadish. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84, 2014.
[3] H. Li, C.-Y. Chan, and D. Maier. Query from examples: An iterative, data-driven approach to query construction. *Proceedings of the VLDB Endowment*, 8(13):2158–2169, 2015.
[4] A.-M. Popescu, O. Etzioni, and H. Kautz. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th international conference on Intelligent user interfaces*, pages 149–157. ACM, 2003.
[5] 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*, pages 2001–2016. ACM, 2015.
[6] L. Qian, M. J. Cafarella, and H. Jagadish. Sample-driven schema mapping. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 73–84. ACM, 2012.
[7] 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. *Proceedings of the VLDB Endowment*, 9(12):1209–1220, 2016.
[8] W. C. Tan, M. Zhang, H. Elmeleegy, and D. Srivastava. Regal+: Reverse engineering spja queries. *Proc. VLDB Endow.*, 11(12):1982–1985, Aug. 2018.
[9] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query reverse engineering. *The VLDB Journal*, 23(5):721–746, Oct. 2014.
[10] C. Wang, A. Cheung, and R. Bodik. 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, pages 452–466, New York, NY, USA, 2017. ACM.
[11] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig. Sqlizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):63, 2017.
[12] T. Yu, M. Yasunaga, K. Yang, R. Zhang, D. Wang, Z. Li, and D. 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*, pages 1653–1663, 2018.
[13] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *EMNLP*, 2018.
[14] M. M. Zloof. Query by example. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 431–438. ACM, 1975.