Maximizing User Domain Expertise to Clarify Oblique Specifications of Relational Queries

by

Christopher James Baik

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2020

Doctoral Committee:

Associate Professor Michael J. Cafarella, Co-Chair
Professor Hosagrahar V. Jagadish, Co-Chair
Associate Professor Kevyn Collins-Thompson
Professor Rada Mihalcea

Christopher James Baik

cjbaik@umich.edu

ORCID iD: 0000-0002-6106-7968

© Christopher James Baik 2020

DEDICATION

Soli Deo gloria. This was not a journey I could not have made on my own.

ACKNOWLEDGMENTS

I am thankful for my advisors, Michael Cafarella and H. V. Jagadish, who guided me through this journey. Mike possesses a visionary mind and imagination that encouraged me to capture a continual enthusiasm for research and innovation. Jag has the ability to listen and pinpoint exactly the right piece of advice needed, whether a birds-eye view truth or a tiny detail, which helped me take exactly the next step I needed. Both have offered their insight and trust to help shape me into a much better scholar.

A thanks to my committee members, Profs. Kevyn Collins-Thompson and Rada Mihalcea, for taking their time to listen, read, and give feedback, as well as my collaborators, Zhongjun Mark Jin and Yunyao Li, for their work on the projects in this dissertation.

A thanks to my former and current officemates in 4929 Bob and Betty Beyster for joining me on walks to the water cooler, including Vaspol Ruamviboonsuk, Joseph Hyunjong Lee, Ayush Goel, Andrew Quinn, Andrew Jinyoung Lee, Muhammed Uluyol, and David Devecsery. To Andrew Jeungahn Lee: thank you for joining me in making the regrettable decision to pay $10 to watch the PhD Movies.

Finally, I am thankful for my family: my wife, Sharon, and her continual encouragement and listening ear; my parents, Kyung Hwan and Myung Ock Baik, who instilled in me from birth that a PhD is my destiny; my sister's family, Amy, Yongwon, Evangeline, and Juliet, who have been a continual source of both support and inspiration; my in-laws, Diane and Jeewon Park, for being my second set of parents; and my church family at Harvest Mission Community Church.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

While there is abundant access to data management technology today, working with data is still challenging for the average user. One common means of manipulating data is with SQL on relational databases, but this requires knowledge of SQL as well as the database's schema and contents. Consequently, previous work has proposed oblique query specification (OQS) methods such as natural language or programming-by-example to allow users to imprecisely specify their query intent. These methods, however, suffer from either low precision or low expressivity and, in addition, produce a list of candidate SQL queries that make it difficult for users to select their final target query.

My thesis is that OQS systems should maximize user domain expertise to triangulate the user's desired query. First, I demonstrate how to leverage previously-issued SQL queries to improve the accuracy of natural language interfaces. Second, I propose a system allowing users to specify a query with both natural language and programming-by-example. Finally, I develop a system where users provide feedback on system-suggested tuples to select a SQL query from a set of candidate queries generated by an OQS system.

CHAPTER 1

Introduction

More than ever before, people today have abundant access to data. Questions that could be only be answered in the past by seeking out an expert can now be handled with a quick command to an AI assistant or a query to a web search engine. Breaking news that could only be heard through word-of-mouth or the next day's newspaper can now be received within seconds through a smartphone notification. New music releases are distributed instantly to customers through streaming outlets rather than through physical media such as CDs or vinyl.

Despite such advances in various consumer-facing technologies, accessing and managing data in database systems, which are commonplace in business and scientific contexts, have remained elusive challenges for non-technical users. In many organizations, it is typical to hire a large band of database administrators and consultants to act as "mediators" between the database system and user. Users specify their data needs to the mediators, who in turn translate the user's specification into a system-friendly representation.

The mediator's task of bridging the gap between the user and the database system is complex. For one, the user's mental model often does not naturally align with the database's logical or physical model of the data. While users think in terms of real-world entities, databases store their contents in various formats to optimize for computation and storage rather than for user comprehension. In addition, when the user lacks understanding of the native representation of the database system, it is possible and even

likely that the user's expressed data need fails to map to a single query in the database's query language. To use an analogy, the English word "love" can be translated to at least four different words in Greek, and a translator must be able to discern the correct word depending on the context. Similarly, it is the mediator's job to convert an ambiguous user expression to a precise query given the context.

Maintaining a large support staff of such mediators can be costly for organizations, as well as cumbersome for users who require a middleman to complete seemingly simple tasks. The goal of my research is to alleviate these costs by *building tools to enable non-technical users to unambiguously specify their data needs to a database system.*

While there are many types of databases such as key-value stores, graph databases, and document-oriented databases, I consider this challenge specifically in the context of relational databases, as they are far and away the most popular type of database to date [64].

1.1 Query Specification Methods

1.1.1 Structured Query Language (SQL)

The most common means of accessing data in a relational database is via Structured Query Language (SQL). SQL is a declarative language, meaning that users need only describe *what* data they want but not *how* to retrieve it programmatically. SQL execution engines are highly optimized to process the queries to retrieve the correct results as fast as possible.

SQL queries are comprised of a series of *clauses*, summarized in Table 1.1. SQL also permits the nesting of queries, allowing the user to issue complex queries to perform a wide variety of operations on tables in a relational database.

While SQL enables the unambiguous specification of complex queries, writing queries is difficult and limits the ability to access data in the hands of a few specialized technical experts. There are several reasons for this.

Clause	Description
`FROM`	Choose tables to perform operations on
`WHERE`	Filter data from tables
`GROUP BY`	Aggregate data
`HAVING`	Filter aggregated data
`SELECT`	Select columns to retrieve
`ORDER BY`	Sort returned data
`LIMIT`	Display top *n* rows

Table 1.1: Main clause types in SQL.

First, *relational schemas are organized for efficiency rather than user comprehension.* Schemas are generally normalized to avoid redundant data storage, splitting real-world entities into multiple relations. Users must manually stitch these relations back together by a series of join operations in the `FROM` clause of a SQL query, but this is a burdensome demand for an untrained user.

Second, *relational operations are challenging for non-experts to master.* While projections in the `SELECT` clause and selections in the `WHERE` clause can be easily understood by most, more complex operations such as aggregates, grouping, and joins are challenging concepts for novices to digest.

Finally, *users often lack knowledge of the database's schema and contents.* Even with prior knowledge of SQL, a user must be able to understand the data within the particular schema they hope to query. In many large enterprises, however, database schemas can become a complex tangled web which the uninitiated user needs to explore for some time before being able to issue queries.

1.1.2 Oblique Query Specification (OQS)

Numerous efforts have been launched to develop more user-friendly interfaces which promote indirect means of specifying structured queries. I dub these systems *oblique query specification (OQS)* systems because they offer an alternative to SQL with oblique (i.e. in-

direct) means of specifying structured queries. The following paragraphs describe some examples of such OQS systems.

1.1.2.1 Keyword Search

Keyword search interfaces emulate web search engines by allowing users to type in keywords to retrieve information. The typical procedure in early work [2, 9, 33] was to discover candidate rows in the database containing each keyword and then to find join paths connecting each combination of candidate rows to construct a result set of joined tuples. These initial systems [2,9,33] were limited to conjunctive select-project-join queries, while later work extended the approach to simple aggregate queries [67], settings where the system does not have a priori access to the database [7], and more complex aggregate queries by augmenting databases with metadata models [10].

1.1.2.2 Natural Language Interfaces

Natural language interfaces enable the user to directly specify their query in human language. Early approaches depended on grammars that were manually-specified [3] or learned from database-specific training examples [27,66], making it difficult to scale them across different database schemas. More recent systems [43,57,61,76,77,79,81] have made great strides in producing database-agnostic interfaces using natural language processing, artificial intelligence, and database techniques.

The typical problem formulation for natural language interfaces is to translate natural language queries into SQL, as a SQL query constitutes an unambiguous specification from a system perspective. This particular problem remains an open challenge as it is difficult, even for human annotators, to translate a potentially ambiguous natural language expression into a single structured query.

1.1.2.3 Programming-By-Example

Programming-by-example (PBE) enables users to describe a query by providing examples with actual or contrived data. The earliest version, Query-by-Example (QBE), asks the user to fill in a skeleton schema [82] with constraints and example output values. Some recent approaches circumvent the need for schema knowledge by considering only project-join queries [36, 58, 59], or by using "abductive" reasoning to select the best selection predicates [25]. Another spin on PBE [73] asks users to provide a sample input database and output rows.

1.1.2.4 Visual Interfaces

Visual interfaces allow the user to specify queries with a visual description involving forms, diagrams, or icons. Catarci et al. [17] provide a survey of such systems, and describe how icon- and form-based systems can support simpler queries and are appropriate for users without domain knowledge, while diagram-based systems generally allow for more complex queries but also require greater expertise from the user.

1.1.3 Overview

The various specification methods described in the previous sections can be categorized along two dimensions: *expressiveness* and *user expertise*. Our goal is to move in the direction of the ideal specification method, which maximizes expressiveness while minimizing user expertise.

1.1.3.1 Expressiveness

Expressiveness describes the complexity of queries supported by a specification method. The expressiveness of a method is generally defined explicitly by the system designer and communicated to the user so they have a clear expectation of what types of queries are

		Technical		Domain
Specification Method	**System(s)**	**Relational Model**	**Schema**	**Factual**
SQL	-	✓	✓	✓
Keyword Search	[2, 9, 10, 33, 67]			
Natural Language	[43, 57, 81]			
Query-by-Example (QBE)	[82]	✓	✓	✓
Exact Project-Join PBE	[59]			✓
Relaxed Project-Join PBE	[36, 58]			
Abductive PBE	[25]			✓
Input-Output PBE	[73]	✓	✓	

Table 1.2: User expertise requirements for various specification methods.

permitted before engaging the system. In practice, the expressiveness of a system is constrained by the level of precision expected of the system. All systems must maintain a relatively high level of precision, as an unreliable system undermines the user's trust and will lead to the system being abandoned. As a result, it is generally the case that specification methods limit the expressiveness to reduce the search space of possible queries to improve overall precision.

1.1.3.2 User Expertise

User expertise can be divided into two major categories: technical and domain expertise. *Technical expertise* indicates a user's proficiency at leveraging technology. In the context of a relational database, this can be further divided into two facets:

1. *Understanding of the relational model.* Does the method require the user to understand complex relational operations such as joins, aggregates, or groupings?

2. *Database schema expertise.* Does the method require the user to know the tables in the database and the relationships between them?

Domain expertise, on the other hand, "defines one's familiarity with a given subject matter; a professional photographer, for instance, has substantial domain expertise in the field of photography" [60]. The query specification process intrinsically requires some

level of domain expertise for the user to understand the terminology within the domain and to effectively evaluate when a task has been completed. Domain expertise, in the context of OQS methods such as PBE, also includes relevant *factual knowledge*, meaning that users possess knowledge of example facts pertaining to their desired query.

We summarize the user expertise requirements for representative systems of various specification methods in Table 1.2.

1.1.4 Challenges

Despite the promise of OQS methods for non-technical users, few, if any, have been widely adopted. According to the Technology Acceptance Model (TAM) [42] for information systems, two factors ultimately decide whether an information system will be accepted by a user: Perceived Usefulness (PU) and Perceived Ease of Use (PEOU). The following challenges facing existing OQS methods are critical to improving each of these factors to enable OQS systems to be adopted by the general public.

Low precision Many existing systems still suffer from low precision, severely crippling Perceived Usefulness. This challenge is perhaps most evident in the case of natural language interfaces. On a benchmark containing SQL queries with joins, aggregates, and nesting, one state-of-the-art natural language interface [79] achieves less than 30% top-1 accuracy. From a user's perspective, it is preferable to seek out an expert to work with than to rely on a system that will only produce the user's expected query 30% of the time.

Low expressivity Other methods, such as PBE, choose to optimize for precision while sacrificing expressivity in the process. Typical formulations of PBE systems requiring low technical expertise constrain the space of queries that can be produced to project-join (PJ) or select-project-join (SPJ) queries, often failing to support queries involving aggregates, or in some cases, even queries with projected numeric columns [25]. Limiting

the permitted expressivity of the system severely limits the Perceived Usefulness of the system.

Target query selection Many existing OQS systems [10,73] produce a set of candidate SQL queries as output. Even if one of these candidate queries is the user's desired query, sifting through the candidate queries requires the user to comprehend SQL to distinguish them, which defeats the very purpose of using the OQS system in the first place and diminishing Perceived Ease of Use. While some systems [21,43,61] support alternate representations of candidate queries, these representations can lack the precision necessary to distinguish two similar candidate queries.

1.2 Dissertation Overview

When a user works with a technical expert (i.e. the database support staff) to pose a query, they can use any and every means possible to supply domain expertise and specify their desired query, whether in natural language, providing a list of hard constraints for the query, offering an expected output tuple, or by drawing the structure of their result table on paper. Synthesizing information from various methods allows the technical expert to clarify the user's specification as needed and to ultimately nail down the user's desired query in the vast search space of candidate queries. Consequently, the ultimate goal is to develop an all-purpose multi-specification system paralleling the ability of the human technical expert.

On the other hand, existing query specification systems largely depend on a single specification method and permanently bind the user to the limitations of the interaction mode at hand. As a result, they require additional clarification to be robust and practically usable.

My thesis is that ***OQS systems should maximize user domain expertise to triangulate the user's desired query***. To this end, I present three approaches to clarify OQS

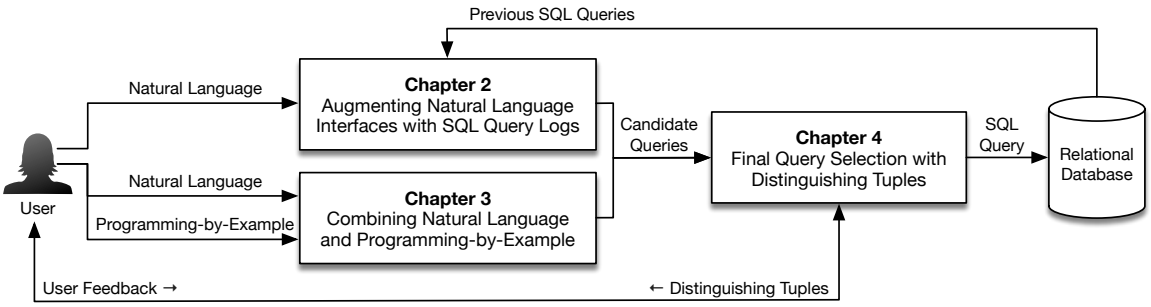


Figure 1.1: An example architecture integrating the approaches in this dissertation.

methods by leveraging user domain expertise to alleviate some of the query specification challenges described in Section 1.1.4. These approaches could be integrated in a single architecture such as in the one shown in Figure 1.1.

The first approach is to use information from previously-issued SQL queries on a database to guide existing OQS systems toward more likely user queries. This is demonstrated in Chapter 2 using a system designed to augment existing natural language interfaces by leveraging insights from a SQL query log, which implicitly contains information on what queries domain experts find interesting. This approach addresses the *low precision* challenge by fluidly combining information from the SQL query log to enable better precision for natural language interfaces.

The second approach is to design system architectures optimized to process multiple specification methods simultaneously. Enabling multiple specification methods essentially provides more possible vectors for users to express their domain expertise, and more information for systems to triangulate the user's desired query. In particular, I present a dual-specification system in Chapter 3 combining natural language and programming-by-example. The system utilizes an architecture designed to maximize the information provided by each of the two modes to address the *low precision* challenge without succumbing to the *low expressivity* challenge.

The final approach is to use domain expertise to tackle the *target query selection* challenge. To that end, I describe a system in Chapter 4 that takes an initial list of candidate

Chapter	Main Research Question	Contributions
Chapter 2: Augmenting Natural Language Interfaces with SQL Query Logs	Can leveraging a SQL query log increase the precision of natural language interfaces?	• the Query Fragment Graph to model the SQL query log • TEMPLAR, a prototype system to augment natural language interfaces • evaluation demonstrates TEMPLAR enables up to 138% increase in top-1 accuracy
Chapter 3: Combining Natural Language and Programming-by-Example	Can a dual-specification OQS outperform a single-specification OQS?	• DUOQUEST, a prototype system implementing a novel dual-specification interaction model • guided partial query enumeration algorithm • studies demonstrating DUOQUEST has better accuracy and expressivity than single-specification systems
Chapter 4: Final Query Selection with Distinguishing Tuples	Can we efficiently use factual domain expertise to help users select their target query?	• a proof that the problem of selecting a target query using tuples is NP-hard • an approximate algorithm to tackle the problem • evaluation demonstrating algorithms can reduce number of tuples used by up to 63%

Table 1.3: A summary of research questions and contributions in this dissertation.

queries generated by any OQS system and generates distinguishing tuples for the user to provide positive or negative feedback on, where positive feedback indicates that a tuple is expected to reside in the desired query's result set. This system tackles the *target query selection* problem by allowing the user to avoid direct interaction with SQL syntax when selecting their desired query.

1.2.1 Outline

The remainder of the dissertation is organized as follows. A summary of research questions and contributions for each chapter is given in Table 1.3.

- In Chapter 2, we consider whether the domain expertise present in a SQL query log can increase the precision of natural language interfaces. We introduce the

Query Fragment Graph as a way to model the SQL query log. We present a system TEMPLAR that augments existing natural language interfaces. We demonstrate the effectiveness of our approach in an experimental evaluation, achieving an up to 138% improvement in top-1 accuracy in existing natural language interfaces by leveraging SQL query log information.

- In Chapter 3, we consider whether using a dual-specification OQS approach can outperform a single-specification approach by eliciting more information from the user's domain expertise. We present a prototype system, DUOQUEST, which leverages a novel dual-specification interaction model and implements an algorithm called guided partial query enumeration to explore the space of possible queries. We present results from user studies and a simulation study that demonstrate significant improvements in accuracy and expressivity over single-specification approaches.
- In Chapter 4, we consider how to tap into users' factual domain expertise by using tuples as a representation for distinguishing candidate queries. We provide a formal definition of the problem of using tuples to select a target query, prove it is NP-hard, develop an approximate algorithm to tackle it, and conducted an experimental evaluation demonstrating that the algorithm can reduce the number of tuples used by up to 63% over other approaches.
- Chapter 5 concludes and describes future work.

CHAPTER 2

Augmenting Natural Language Interfaces with SQL Query Logs[1]

The task of a natural language interface to databases (NLIDB) has been primarily modeled as the problem of translating a natural language query (NLQ) into a SQL query. State-of-the-art systems developed to solve this task take one of two architectural approaches: (1) the *pipeline approach* of converting an NLQ into intermediate representations then mapping these representations to SQL (e.g. [43, 57, 61, 77]), and (2) the *deep learning approach* of using an end-to-end neural network to perform the translation (e.g. [71, 76, 81]).

However, as pointed out by [45], one fundamental challenge in supporting NLIDBs is *bridging the semantic gap* between a NLQ and the underlying data. When translating NLQ to SQL, this challenge arises in two specific problems: (1) *keyword mapping* and (2) *join path inference*. *Keyword mapping* is the task of mapping individual keywords in the original NLQ to database elements (such as relations, attributes or values). It is a challenging task because of the ambiguity in mapping the user's mental model and diction to the schema definition and contents of the database. *Join path inference* is the process of selecting the relations and join conditions in the `FROM` clause of the final SQL query, and is difficult because NLIDB users do not have a knowledge of the database schema or

[1] ©2019 IEEE. Reprinted, with permission, from Christopher Baik, H. V. Jagadish, Yunyao Li, *Bridging the Semantic Gap with SQL Query Logs in Natural Language Interfaces to Databases*, 35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019, https://doi.org/10.1109/ICDE.2019.00041

System	NLQ Preprocess	Rel/Attr Mapping	Value Mapping	Join Path Inference	SQL Post-process
Precise [57]	Tokenizer + Charniak [18] parser	WordNet [52]	Same as rel/attr	Max-flow algorithm + User interaction	N/A
NaLIR [43]	Stanford Parser [20]	WordNet [52] + User interaction	Same as rel/attr	Preset path weights + User interaction	Query tree heuristics + User interaction
SQLizer [77]	Sempre [6]	word2vec [51]	Same as rel/attr	Hand-written repair rules	Hand-written repair rules
ATHENA [61]	Tokenizer	Synonym lexicon + Pre-defined ontology	Index with semantic variants	Pre-defined ontology	N/A
Seq2SQL [81]	Tokenizer + Stanford CoreNLP [48]	GloVe [56] + character n-grams [32]	Unsupported	N/A	N/A
SQLNet [76]	Tokenizer + Stanford CoreNLP [48]	GloVe [56]	Unsupported	N/A	N/A
DBPal [71]	Replace literals with placeholders	Unspecified	word2vec [51]	Select min-length path	SQL syntax repair + Fill placeholders

Table 2.1: State-of-the-art NLIDBs. Upper half are pipeline-based, lower half are end-to-end deep learning systems.

SQL and therefore cannot explicitly specify the intermediate tables and joins needed to construct a final SQL query.

Table 2.1 summarizes several state-of-the-art systems and their strategy to handle each step of NLQ to SQL translation. The upper half lists pipeline-based systems, where each subproblem is explicitly handled, while the lower half are deep learning systems which implicitly tackle these challenges by the choice of input representation and network architecture. The *keyword mapping* task is split into the *Rel/Attr Mapping* and *Value Mapping* columns because some systems have independent procedures for handling each. Some common patterns emerge:

- For *keyword mapping*, the vast majority of systems make use of a lexical database such as WordNet [52] or a word embedding model [51, 56].
- *Join path inference* is primarily handled via user interaction [43] or heuristics such as selecting the shortest join path [71] or hand-written repair rules [77].

While each of these approaches works reasonably well, there is still significant room for improvement. For example:

Example 2.1. *John issues an NLQ: "Find papers in the Databases domain" on an academic database (Figure 2.1) using a pipeline NLIDB. John's intended SQL query is:*

```
SELECT p.title
FROM publication p, publication_keyword pk, keyword k,
  domain_keyword dk, domain d
WHERE d.name = `Databases' AND p.pid = pk.pid
  AND k.kid = pk.kid AND dk.kid = k.kid AND dk.did = d.did
```

The NLIDB attempts ***keyword mapping*** *by matching "papers" in the NLQ to either the relation* `publication` *or* `journal`*, and "Databases" to a value in the* `domain` *relation. It maps "papers" to* `journal` *because they have a high similarity score in the NLIDB's word embedding model. After this, the NLIDB performs* ***join path inference*** *by examining the schema graph and selects the shortest join path from* `journal` *to* `domain` *to form the (unintended) SQL query:*

```
SELECT j.name
FROM journal j, domain_journal o, domain d
WHERE d.name = `Databases' AND j.jid = o.jid
  AND o.did = d.did
```

The example demonstrates how error in keyword mapping can propagate through the pipeline to produce an incorrect SQL query. Even when the keyword mapping is correct, however, the join path inference remains as a challenge:

Example 2.2. *In the keyword mapping process for John's NLQ, assume the NLIDB correctly matched "papers" to* `publication`*. The NLIDB examines the schema graph and its algorithm selects the shortest path from* `publication` *to* `domain`*. The returned SQL does not match John's intent:*

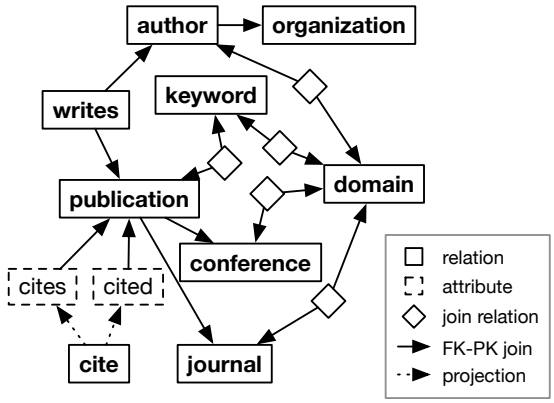


Figure 2.1: A simplified version of the Microsoft Academic Search database's schema graph.

```
SELECT p.title
FROM publication p, conference c,
  domain_conference dc, domain d
WHERE d.name = `Databases' AND p.pid = c.pid
  AND c.cid = dc.cid AND dc.did = d.did
```

While there is always inherent ambiguity introduced in NLQs that even humans have difficulty interpreting, our goal is to improve the accuracy of *keyword mapping* and *join path inference* in NLIDBs to better match the user's intent.

Recent end-to-end deep learning systems [11, 30, 71, 76, 79, 81] show the great promise of learning from large volumes of NLQ-SQL pairs. However, manually creating labeled NLQ-SQL pairs is still costly and time-consuming. Despite recent efforts to synthesize NLQ-SQL pairs [34, 71, 80] or derive them from user descriptions of SQL queries [14], obtaining realistic labeled data remains an open research challenge.

Our Approach While NLQ-SQL pairs are rarely available in large quantities for a given schema, large SQL query logs are more readily available given that NLIDBs are often built not for freshly instantiated databases, but for existing production databases [26, 31]. Although the SQL query log is not a typical supervised learning training set of input-

output pairs, an output set of rich data like the SQL query log can still provide value in translation, akin to the way that one could infer much about what is being communicated and what should be spoken next even by listening to only one end of a phone conversation. Our approach is to use the information in the SQL query log of a database to select more likely keyword mappings and join paths for SQL translations of NLQs.

We propose a system TEMPLAR, which augments existing pipeline-based NLIDBs such as [43, 57, 61, 77] with SQL query log information. While it is also possible to augment end-to-end deep learning NLIDBs, this would require additional pre- or post-processing, and we leave it for future work. Consider the user of TEMPLAR with our running example:

Example 2.3. *John issues the NLQ from Example 2.1 on a NLIDB augmented with* TEMPLAR. *The NLIDB defers the* ***keyword mapping*** *to* TEMPLAR*, which uses information in the SQL query log to determine* `publication` *as the most likely mapping. The NLIDB receives this information, performs any necessary processing, and then defers* ***join path inference*** *to* TEMPLAR *by passing the mapped relations and attributes to it.* TEMPLAR *takes the input and again uses the SQL query log to conclude that the most likely join path involves connecting* `publication` *to* `domain` *via the* `keyword` *relation. This join path is passed back to the NLIDB, which constructs the final SQL query matching John's intent.*

Technical Challenges Unlike traditional learning tasks where full input-output pairs (i.e. NLQ-SQL pairs) are used to train a model, we use only output logs (i.e. SQL queries). Consequently, the information in the SQL query log does not directly map to the translation task. Furthermore, even with large query logs, it is likely that most queries are not exact repeats of queries previously issued. Finally, our goal is to augment, rather than replace, NLIDBs, so we need TEMPLAR to be able to assist multiple NLIDBs through a simple common interface. In short, the challenges are to (1) *selectively activate information in the SQL log* for NLQ-SQL translation, (2) *allow the generation of new SQL queries* not in the log, and (3) *gracefully integrate log information with existing techniques* in NLIDBs.

Contributions Our main contributions are as follows:

- We propose the *query fragment* as an atomic building block for SQL, providing a fine-grained view of a SQL log to allow *selective activation of information in the log*. Query fragments can be mixed and matched to *allow the generation of new SQL queries* not yet observed in the query log.
- We propose the *Query Fragment Graph* as a novel abstraction to enhance the accuracy of keyword mapping and join path inference in NLIDBs by modeling the co-occurrence of query fragments from a SQL query log, and *gracefully integrating this with existing techniques* to improve the accuracy of keyword mapping and join path inference in NLIDBs.
- We introduce a prototype system Templar, which augments existing NLIDBs without altering their internal architecture.
- We demonstrate by an extensive evaluation on how Templar can improve the top-1 accuracy of state-of-the-art NLIDBs by up to 138% on our benchmarks.

Organization We discuss related work (Section 2.1), then present the architecture of Templar and formal problem definitions (Section 2.2), before introducing the query fragment and Query Fragment Graph to model the SQL query log (Section 2.3). We then explain our algorithms for improving the accuracy of keyword mapping and join path inference by leveraging the Query Fragment Graph (Sections 2.4-2.5). We present our experimental evaluation of NLIDBs augmented with Templar (Section 2.6), and summarize (Section 2.7).

2.1 Related Work

Our work builds upon multiple streams of prior work:

Natural language interfaces to databases (NLIDB) Research on NLIDBs extends as far back to the sixties and seventies [3], when interfaces were focused on solutions tailored to a specific domain. Early approaches depended on grammars that were manually-specified [3] or learned from database-specific training examples [27, 66], making it difficult to scale them across different database schemas.

Since then, advances in deep learning have inspired efforts to build an end-to-end deep learning framework to handle natural language queries [23, 47, 78]. The limiting factor for such systems is the need for a large set of NLQ to SQL pairs for each schema, and consequently some work focuses on the challenge of synthesizing and collecting NLQ-SQL pairs [14, 34, 71, 80] to be able to train these systems. Some deep learning-based end-to-end systems [71, 76, 81] make use of the sequence-to-sequence architecture, and these systems can benefit from the enhancements Templar provides to keyword mapping, but not from join path inference because their application is confined to single-table schemas. More recent syntax tree-based systems [30, 79] handle join path inference as a separate step in the query inference process and can take advantage of our contributions for both keyword mapping and join path inference.

An alternative approach has been to combine techniques from the natural language processing and database communities to construct pipeline-based NLIDBs. Such systems often utilize intermediate representations in the NLQ to SQL translation process, such as a parse tree [43], query sketch [70,77], or an ontology [61]. They also ensure reliability by doing at least one of the following: explicitly defining their semantic coverage [43, 46, 57], allowing the user to correct ambiguities [43], asking the user to provide a mapping from a database schema to an ontology [61], or by engaging in an automated query repair process [77]. Templar can enhance the performance of these NLIDBs by leveraging query logs as an additional data source to increase accuracy.

Keyword search Keyword search interfaces [2,8–10,33,67] emulate web search engines by allowing users to type in keywords to retrieve information. These keyword search interfaces often face the keyword mapping and join path inference problems that were described in our work, but Templar is the first to make use of the SQL query log to address these issues.

Using query logs Previous work used SQL query logs to autocomplete SQL queries [38], proposing a similar abstraction to query fragments for a different purpose. QueRIE [24] and qunits [53] organized the query log in a similar fashion to the Query Fragment Graph, but for the purposes of query recommendations and keyword queries, respectively.

2.2 Overview

2.2.1 Preliminaries

We first introduce some preliminary definitions.

The schema graph depicts the relations and their connections in a relational database:

Definition 2.1. *A **schema graph** is a directed graph $G_s = (V, E, w)$ for a database D with the following properties:*

- *V consists of two types of vertices:*
  - *Relation vertices $V_R \subseteq V$, each corresponding to a relation in D.*
  - *Attribute vertices $V_\alpha \subset V$, each corresponding to an attribute in D.*
- *E consists of two types of edges:*
  - *Projection edges $E_\pi \subseteq E$, each extending from a given relation vertex to each of its corresponding attribute vertices.*

- *FK-PK join edges $E_{\bowtie} \subset E$, each extending from each foreign key attribute vertex to its corresponding primary key attribute vertex.*

- $w : V \times V \longrightarrow [0,1]$ *is a function that assigns a weight to each pair of vertices which have an edge in E.*

A join path is a specific type of tree within the schema graph, which can be represented by a combination of relations and join conditions in a SQL query:

Definition 2.2. *Given a schema graph G_s and a bag of relations B_R, a* ***join path*** (V_j, E_j, V_t) *is a tree of vertices $V_j \subset G_s$ and edges $E_j \subset G_s$ spanning all terminal vertices $V_t \subset G_s$, where each relation instance in B_R is represented by a terminal vertex $v_R \in V_t$.*

2.2.2 Definitions

As we will discuss in detail in Section 2.3, a complete SQL query is too large and too specific a unit of data to be able to use it effectively to represent a SQL log. Instead, we use *query fragments*, which are pieces of SQL queries:

Definition 2.3. *A* ***query fragment*** $c = (\chi, \tau)$ *is a pair of:*

- *χ: a SQL expression or non-join condition predicate;*

- *τ: the context clause in which χ resides.*

For example, in the SQL query:

```
SELECT t.a FROM table1 t, table2 u
WHERE t.b = 15 AND t.id = u.id
```

The query fragments are (`t.a`, `SELECT`), (`table1`, `FROM`), (`table2`, `FROM`), (`t.b = 15`, `WHERE`).

Keyword phrases in a NLQ are mapped to query fragments by NLIDBs to form *query fragment mappings*:

Definition 2.4. *A **query fragment mapping** $m = (s, c, \sigma)$ is a triple of a keyword s, a query fragment c, and a similarity score σ between the keyword and query fragment.*

A selection of mappings for an NLQ form a *configuration*:

Definition 2.5. *A **configuration** $\phi(S)$ of a set of keywords S is a selection of exactly one query fragment mapping (s_k, c_k, σ_k) for each keyword $s_k \in S$, where c_k is a query fragment, and σ_k is the associated similarity score for the keyword and fragment.*

2.2.3 Problem Definitions

We now present a formal definition for the *keyword mapping* and *join path inference* problems.

2.2.3.1 Keyword Mapping

The keyword mapping problem is described by the function:

$$\Phi = \textsc{MapKeywords}(D, S, M)$$

The input to the problem is a database D, a set of keywords representing an NLQ, $S = \{s_1, s_2, ..., s_n\}$, where each keyword $s_k \in S$ can be comprised of multiple words or tokens in natural language; and a set of metadata annotations, M, where each element $M_k = (\tau_k, \omega_k, \mathcal{F}_k, g_k)$ of M includes parser metadata about s_k: the context τ_k of the query fragment that should be mapped to s_k, an optional predicate comparison operator ω_k, an optional ordered list of aggregation functions $\mathcal{F}_k$, and a boolean g_k which if true, indicates that the resulting mapping of s_k should be grouped. The goal of the problem is to return a list of *configurations* Φ ordered by likelihood.

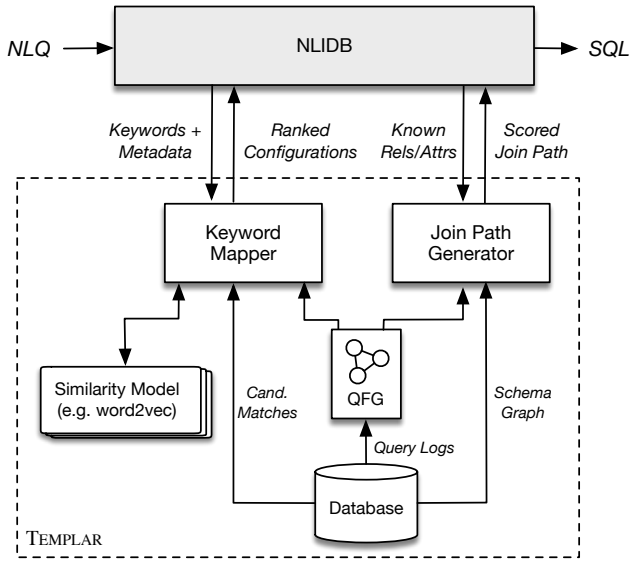


Figure 2.2: The overall architecture of an NLIDB augmented with TEMPLAR.

2.2.3.2 Join Path Inference

The join path inference problem is described by the function:

$$J = \textsc{InferJoins}(G_s, B_D)$$

The input is a schema graph G_s, a bag (i.e. a multiset) of attributes and relations B_D that are known to be part of the SQL query. The goal is to return a list of join paths J on G_s ranked from most to least likely.

2.2.4 Architecture

TEMPLAR's architecture is shown in Figure 2.2. It interfaces with the NLIDB it is augmenting on two fronts: one for keyword mapping, and the other for join path inference.

The ***Keyword Mapper*** carries out the execution of MAPKEYWORDS, and uses a word similarity model such as word2vec [51] or GloVe [56], the query fragment graph (QFG) which stores the SQL query log information, and the database itself to retrieve candidate matches. The ***Join Path Generator*** executes INFERJOINS, and it utilizes the QFG and the

schema graph of the database to infer join paths.

2.2.5 NLIDB Prerequisites

An NLIDB to which we can apply our approach is responsible for the following:

- It must be able to parse the NLQ into keywords, which may require recognition of multi-word entities. Each keyword should have associated metadata (query fragment type, predicate operator, aggregation functions, and presence of a group-by) for the keyword mapping problem.
- It is responsible for constructing a SQL query given the keyword mappings and join paths provided by Templar.

The categories of metadata we expect as input in MapKeywords are all obtainable using existing parser technology [39, 54, 62] by existing NLIDBs [43, 77].

Since the two main interface calls of keyword mapping and join path inference are independent of one another in our approach, we do not enforce any ordering of when and how these calls should be made within the NLIDB. However, in every currently known system in Table 2.1, the keyword mapping step precedes the join path inference step.

The interface to pipeline-based NLIDBs such as [43, 57, 61, 77] is transparent, as most already support the above requirements or can be easily modified to do so. Integrating Templar into an end-to-end deep learning NLIDB is possible by integrating the information from the SQL query log into the input representation or by performing some pre-processing and/or post-processing, but we leave this for future work.

2.2.6 Example Execution

In this section, we describe an example execution of a generic pipeline-based NLIDB augmented with Templar. Consider the architecture in Figure 2.2 and the following example NLQ from the Microsoft Academic Search (MAS) dataset [43] with schema in Figure 2.1:

Example 2.4. *Return the papers after 2000.*

First, the NLIDB parses the NLQ to return the keywords which map to elements in the database and corresponding parser metadata. In Example 2.4, the keywords emitted by the NLIDB would be *papers* and *after 2000*. NLIDBs have various techniques of producing the metadata, whether through semantic parsing [77] or a designated lexicon of keywords [43]. The NLIDB in [43] would return that *papers* is in the `SELECT` context because it is a direct child of the keyword *Return* in the parse tree, and *after 2000* would be in the `WHERE` context because *after* is a reserved keyword corresponding to the predicate comparison operator > in the NLIDB's lexicon.

The keywords are passed to the ***Keyword Mapper***, which maps each keyword to candidate query fragments using the keyword metadata and information about the database schema and contents. These candidate query fragment mappings are individually scored using a similarity model (such as word2vec [51]) and information from the *Query Fragment Graph (QFG)*. For Example 2.4, the candidate mappings for *papers* includes (`journal.name`, `SELECT`) and (`publication.title`, `SELECT`), and *after 2000* is mapped to (`publication.year > 2000`, `WHERE`).

A configuration is generated by selecting one candidate mapping per keyword. The top-κ most likely candidate configurations are returned by the ***Keyword Mapper***. Example 2.4 produces at least two candidate configurations, whose mapped query fragments, respectively, are:

- `[(journal.name, SELECT);`
  `(publication.year > 2000, WHERE)]`
- `[(publication.title, SELECT);`
  `(publication.year > 2000, WHERE)]`

These configurations are then sent back to the NLIDB, which can augment the ranked configurations with other information such as domain-specific knowledge.

After processing the configurations, the NLIDB sends known relations for each candidate SQL translation to the ***Join Path Generator***, which identifies the most likely join path and returns it along with an associated score.

For the schema graph shown in Figure 2.1 and continuing with Example 2.4, this step will produce the join path `journal-publication` for our first configuration, and the single relation `publication` for the second.

Finally, it is the NLIDB's responsibility to construct the final SQL query and return it. Any post-processing, such as the hand-written repair rules in [77] or soliciting additional user interaction as in [43] may also be performed at this point. For our running example, the final SQL queries returned by the NLIDB for each candidate configuration would be:

- ```
  SELECT j.name FROM journal j, publication p
  WHERE p.year > 2000 AND j.jid = p.jid
  ```
- ```
  SELECT title FROM publication WHERE year > 2000
  ```

2.3 Query Log Model

In this section, we explore how to model information in the SQL query log to aid in NLQ to SQL translation. Consider the SQL query log in Figure 2.3a and the example task:

Example 2.5. *The task is:*

- ***NLQ**: Select all papers from TKDE after 1995.*
- ***SQL**:*
  ```
  SELECT p.title FROM journal j, publication p
  WHERE j.name = 'TKDE'
  AND p.year > 1995 AND j.jid = p.jid
  ```

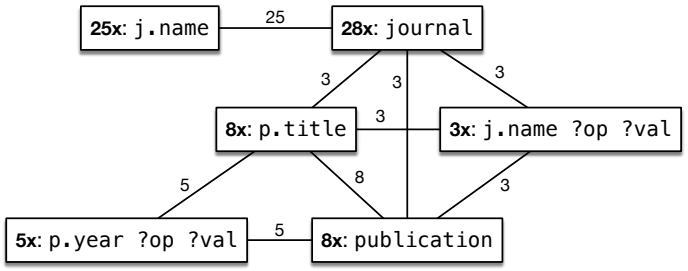
First, we want *to generate queries not yet observed in the SQL log*—i.e. not be constrained to only translate to queries already in the log. In Example 2.5, the NLQ has the keyword

```
25x: SELECT j.name FROM journal j
5x: SELECT p.title FROM publication p WHERE p.year > 2003
3x: SELECT p.title FROM journal j, publication p WHERE j.name = 'TMC'
      AND p.pid = j.pid
```

(a) Example query log.

25x: j.name	**28x**: journal	**5x**: p.year ?op ?val
8x: p.title	**8x**: publication	**3x**: j.name ?op ?val

(b) Query fragment occurrences.



(c) Query fragment graph.

Figure 2.3: Storing query log information in the QFG.

papers which might map to `publication.title` or `journal.name`. If we are limited to selecting existing SQL queries in the log to translate to, the NLQ could erroneously be translated to `SELECT j.name FROM journal j`.

To avoid this, we break down SQL queries into query fragments which can be mixed and matched to form new SQL queries, and count occurrences of each query fragment in the log as in Figure 2.3b.

Now, consider that we boost the scores of commonly-occurring query fragments in the SQL log. Unfortunately, there is still a high chance that "papers" will be mapped to `journal` because of its high frequency in the log.

Consequently, we want to *selectively activate information in the log* only when helpful for the NLQ at hand. The intuition is that the full NLQ provides context for each individual keyword, and this should be leveraged to illuminate what queries in the SQL log are relevant to the NLQ. In Example 2.5, the keywords are *papers*, *TKDE*, and *after 1995*. A human expert would that *TKDE* is referring to a journal and *after 1995* refers to a year,

and can conclude that *papers* isn't referring to `journal.name` because the NLQ would be redundantly asking for "all journals from a journal".

Finally, we want to *maximize the semantic information in the SQL query log*. We observe a distinction between the more abstract semantic information and the specific value instances in a query. For Example 2.5, we can replace specific values in the NLQ with placeholders: *Select all papers from (journal) after (year)*, preserving the semantic structure while obscuring exact values. Similarly, we can put placeholders in the SQL:

```
SELECT p.title
FROM journal j, publication p
WHERE j.name ?op ?val AND p.year ?op ?val
 AND j.jid = p.jid
```

Using such placeholders allows us to focus on the recurrence of semantic contexts without being distracted by specific values. Consequently, it allows us to make more extensive use of the data in the SQL query log as more query fragments in the log are likely to match any given keyword in a NLQ.

We implement three levels of *obscurity* for query fragments. The first level, *Full*, retains all values in the original query. The second, *NoConst*, replaces literal constants with a placeholder to convert a fragment `p.year > 2000` into `p.year > ?val`. Finally, we further obscure comparison operators in *NoConstOp* to make the fragment `p.year ?op ?val`.

2.3.1 Query Fragment Graph

While automated NLIDBs don't have the benefit of human logic, the SQL query log can play a similar role by using the full context of a NLQ to revise individual keyword mappings. Previous user queries in the log in Figure 2.3a show that years are often queried in the context of `publication.title`, and similarly, when a specific journal name such

as *TMC* is a predicate in a query, the user is often querying `publication.title`. This observation leads us to desire not only the occurrences of individual query fragments in the SQL log, but also the co-occurrences of query fragments—in other words, given the information in the SQL log, when one query fragment appears in a query, how likely is it that another query fragment is present in the query?

Given the intuition above, we introduce the ***Query Fragment Graph (QFG)*** as a data structure to store the information in a SQL query log.

Definition 2.6. *A **query fragment graph** for database D and SQL query log L is a graph $G_f = (V_f, E_f, n_v, n_e)$ where:*

1. *each vertex $v \in V_f$ represents a query fragment in L;*
2. *each edge $e \in E_f$ exists if and only if two query fragments co-occur in L;*
3. *$n_v : V_f \longrightarrow \mathbb{Z}_{\geq 0}$ is a function which maps V_f to the number of occurrences in L of the query fragment represented by each $v \in V_f$;*
4. *$n_e : V_f \times V_f \longrightarrow \mathbb{Z}_{\geq 0}$ is a function which maps each pair of vertices to the co-occurrence frequency in L of the two query fragments represented by the vertices.*

In short, the QFG stores information on *query fragment occurrences* (n_v) in the log, as well as *co-occurrence relationships* (n_e) between each pair of query fragments.

2.4 Keyword Mapping

In this section, we explain the keyword mapping procedure. While many techniques described here are already applied in existing work, we explain each step in detail to keep this work self-contained, and to clearly show how our novel approach of using SQL query log information comes into play.

Mapping keywords involves three steps: (1) retrieving candidate keyword to query fragment mappings, (2) scoring and retaining the top-κ candidates, and (3) generating

Algorithm 1 Mapping Keywords

1: **function** MapKeywords(D,S,M)
2: $\mathcal{R} \leftarrow \{\}$
3: **for** $k \leftarrow 1, \ldots, |S|$ **do**
4: $(\tau_k, \omega_k, \mathcal{F}_k, g_k) \leftarrow M_k$
5: $C_k \leftarrow$ KeywordCands($D, s_k, \tau_k, \omega_k, \mathcal{F}_k, g_k$)
6: $R_k \leftarrow$ ScoreAndPrune(s_k, C_k, κ)
7: $\mathcal{R}.add(R_k)$
8: Φ = *genAndScoreConfigs*($\mathcal{R}$)
9: **return** Φ

and scoring configurations. Information from the query fragment graph is used in the final step to score configurations according to the evidence in the SQL query log.

We now describe our algorithm for the MapKeywords function, shown in Algorithm 1. We loop through all the keywords $s_k \in S$ with their corresponding metadata, then combine and rank them to form our output configurations.

2.4.1 Retrieving Candidate Mappings

The function KeywordCands in Algorithm 2 maps a keyword s, along with its associated metadata $(\tau, \omega, \mathcal{F}, g)$, to its candidate mappings C by querying the database D.

First, we evaluate whether s contains a number (Line 3), such as in the keyword *after 2000*. If so, we return all numeric attributes in the database that match a predicate formed by the number extracted from s with the operator ω for s (Line 5). For the keyword *after 2000*, we return all attributes containing at least one value that satisfies the predicate `?attr > 2000`. Predicates are constructed from matching attributes and added to the candidate set C.

If s does not contain a number, we have three different cases. In the first two cases, where the context τ of the query fragment is `FROM` or `SELECT`, we simply add either all the relations or all the attributes (along with relevant metadata) of D to the candidate set C.

For the final case covering all other structures, we first run a full-text search with every Porter-stemmed [72] whitespace-separated token in s to retrieve all matching text

Algorithm 2 Retrieve Candidate Keyword Mappings

```
1:  function KeywordCands(D,s,τ,ω,F,g)
2:      C ← {}
3:      if containsNumber(s) then
4:          s_num ← extractNumber(s)
5:          β ← findNumericAttrs(s_num, ω)
6:          for b ∈ β do
7:              C.add((Pred(b, ω, s_num), WHERE))
8:      else
9:          if τ = FROM then
10:             for r ∈ getRelations(D) do
11:                 C.add((r, τ))
12:         else if τ = SELECT then
13:             for α ∈ getAttributes(D) do
14:                 C.add((Attr(α, F, g), τ))
15:         else
16:             for t ∈ findTextAttrs(s) do
17:                 C.add((Pred(t, =, s), WHERE))
18:     return C
```

attributes T in D (*findTextAttrs* in Line 16). For example, for the keyword *restaurant businesses*, the stemming procedure would result in the tokens *restaur busi*, and we run the following SQL query, replacing `?attr` with each text attribute in D:

```
SELECT DISTINCT(?attr) FROM ?rel
WHERE MATCH(?attr)
  AGAINST (`+restaur* +busi*' IN BOOLEAN MODE)
```

If any of the stemmed tokens from s exactly match the stemmed attribute or relation names of a candidate query fragment, we remove them so as not to unnecessarily constrain our search. For example, if the keyword is *movie Saving Private Ryan* and a candidate query fragment mapping is an attribute from the `movie` relation, we remove the token *movie* from our full-text search query when searching on that attribute. For each matching text attribute, we then construct a predicate for the `WHERE` context.

Algorithm 3 Score and Prune Keyword Mappings

```
1:  function ScoreAndPrune(s,C,κ)
2:      R ← {}
3:      for c ∈ C do
4:          if containsNumber(s) then
5:              s_num ← extractNumber(s)
6:              s_text ← s − s_num
7:              σ ← sim_num(s_text, c)
8:          else
9:              σ ← sim_text(s, c)
10:         R.add((s, c, σ))
11:     sort R by descending σ
12:     return Prune(R, κ)
```

2.4.2 Scoring and Pruning

Our next step is to retain only the top-κ most likely mappings from C with the function ScoreAndPrune.

We calculate a score σ for each keyword mapping in the range $[0, 1]$. For comparing keywords with purely text tokens against relation and attribute names and text predicates, we can use a similarity function sim_{text} (Line 9) through a word embedding model such as word2vec [51] or GloVe [56]. For keywords including numeric tokens, we execute (i.e. $exec(c)$) the candidate predicate on the database, then evaluate the similarity of only the text tokens if the predicate returns a non-empty set, and return a small ϵ value otherwise:

$$sim_{num}(s_{text}, c) = \begin{cases} sim_{text}(s_{text}, c), & \text{if } exec(c) \nrightarrow \emptyset \\ \epsilon, & \text{otherwise} \end{cases}$$

σ is then combined into a tuple with the original keyword s and candidate mapping c and added to the result set R, which is finally sorted by descending σ score. We then prune R to prevent a combinatorial explosion when generating configurations, using the following Prune procedure (Line 12):

- If there are any candidates in R that are exact matches ($\sigma \geq 1 - \epsilon$ for a small ϵ), we

prune away all remaining non-exact candidates.

- Otherwise, we prune R to the top-κ results, including any results that have a non-zero σ value that is equal to the σ of the candidate at the κ-th place.

2.4.3 Ranking Configurations

At this point, we have a set of candidate mappings for each keyword $s_k \in S$. We combine and score them (Line 8 of Algorithm 1) to form candidate configurations for S. We first describe a standard way of scoring configurations, then show how we can apply the SQL query log to improve scoring.

2.4.3.1 Word Similarity-Based Score

A naïve scoring function for configurations selects the best mapping for each keyword independently. We can take the geometric mean of the scores of all mappings to accomplish this:

$$Score_{\sigma}(\phi) = [\prod_{(s_k, c_k, \sigma_k) \in \phi} \sigma_k]^{\frac{1}{|\phi|}}$$

We prefer the geometric mean over the arithmetic mean, as in [77], to mitigate the impact of the variation in ranges of values for each keyword's candidate mapping scores.

2.4.3.2 Query Log-Driven Score

Since we have the query log information available to us via the Query Fragment Graph, we leverage this information to derive an improved scoring function contextualized for our specific database schema.

While word similarity-based scoring considers each mapping independently, we now consider the *collective score* of each configuration of mappings. Previous work such as [43] attempts a collective scoring approach based on *mutual relevance* which considers the proximity of keywords in the natural language dependency tree in relation to the edge

weights connecting the candidate query fragments within the schema graph. Unfortunately, these schema graph edge weights are assigned manually without justification.

In contrast, the intuition behind our collective scoring mechanism is to give a higher score to configurations containing query fragments that frequently co-occur in queries in the SQL query log. Instead of relying on the system administrator's ability to preset the schema graph edge weights to match an anticipated workload, we derive our scoring directly from previous users' queries in the SQL query log.

To accomplish this, we calculate a metric for the co-occurrence of pairs of query fragments in the QFG, then aggregate this metric, along with the previously-computed similarity scores, over all query fragments in the configuration to derive a final score. We use the Dice similarity coefficient [29] to reflect the co-occurrence of two query fragments c_1 and c_2 in the QFG, defined as follows:

$$Dice(c_1, c_2) = \frac{2 \times n_e(c_1, c_2)}{n_v(c_1) + n_v(c_2)}$$

We accumulate *Dice* for every pair of non-relation (i.e. not in the `FROM` context) fragments $(c_1, c_2) \in \phi_{\tau \neq \texttt{FROM}} \times \phi_{\tau \neq \texttt{FROM}}$:

$$Score_{QFG}(\phi) = [\prod_{(c_1,c_2) \in \phi^2_{\tau \neq \texttt{FROM}}} Dice(c_1, c_2)]^{\frac{1}{|\phi|}}$$

The query fragments in the `FROM` context are excluded because involving relations can add information skewing the aggregate score—e.g. if `journal.name` is in a SQL query, then the relation `journal` is required to be by the rules of SQL, adding unnecessary redundancy to the aggregated Dice score. In addition, relations in the `FROM` clause are explicitly handled by our join path inference procedure, so we defer the evaluation of these query fragments for later.

Finally, we perform a linear combination (governed by a parameter $\lambda \in [0, 1]$) of $Score_\sigma$

and the query log-driven score $Score_{QFG}$ to produce a final configuration score:

$$Score(\phi) = \lambda Score_{\sigma}(\phi) + (1 - \lambda) Score_{QFG}(\phi)$$

We can also replace this means of combining evidence from multiple sources with other approaches, such as the Dempster Shafer Theory in [8]. We opt for a linear combination due to its simplicity and because it works sufficiently well in practice.

All configurations are now scored using $Score(\phi)$, ranked by descending score, and returned by MapKeywords.

2.5 Join Path Inference

In this section, we describe how we generate join paths for a set of attributes and relations selected to be part of the final SQL query by the keyword mapping procedure, and show how we use the SQL query log to improve this process.

Example 2.6. *Consider that the NLIDB selected the following query fragments to be part of a SQL query of the schema given in Figure 2.1:*

- *(*`publication.title`*,* `SELECT`*)*
- *(*`domain.name = 'Databases'`*,* `WHERE`*)*

InferJoins should output the desired join path:

```
publication - publication_keyword - keyword
  - domain_keyword - domain
```

2.5.1 Generating Join Paths

The process of generating the set of optimal join paths from a set of known relations B_R and a schema graph G_s has previously been modeled as the Steiner tree problem [41],

where the goal is to find a tree on a graph that spans a given set of vertices with minimal total weight on its edges.

The expected input to the ***Join Path Generator*** is a bag of the attributes and relations B_D already known to be in the desired SQL translation. B_D can be converted to the bag of known relations B_R simply by replacing each attribute with its parent relation in G_s.

We use a known algorithm [41] for solving Steiner trees to find the set of optimal join paths for any given configuration. These optimal join paths, however, change depending on how weights are assigned to edges in the schema graph. We outline two ways to do this, first without information from the query log, and then adding in query log information.

2.5.1.1 Default Edge Weights

The default weight function w for edges in the schema graph is to assign every edge a weight of 1. If we solve the Steiner tree problem with this weight function, we are essentially finding join paths with the minimal number of join edges that span all the known relations.

For Example 2.6, this approach will produce the shortest join path between `publication` and `domain`, which is either:

- `publication–conference–domain_conference–domain`
- `publication–journal–domain_journal–domain`

Neither of these join paths are the one desired by the user.

2.5.1.2 Query Log-Driven Edge Weights

We look to the query log to provide some grounding for generating join paths. In contrast to previous work which depends on the system administrator to set schema graph edge weights [43], on hand-written repair rules [77], or a predefined ontology [61], query log

information is driven by actual user queries executed on the system. Query log information allows us to prefer commonly queried join paths, even if they are longer, and also mitigates the number of situations where there are identical scores given to equal-length join paths.

We leverage the co-occurrence values of relations in the QFG to adjust the weights on the schema graph. Given any two vertices $(v_1, v_2) \in G_s$, and the function $q : V \longrightarrow V_{QF}$ which maps a vertex in the schema graph G_s to its corresponding vertex in the QFG, the new weight function is:

$$w_L(v_1, v_2) = \begin{cases} 1 - Dice(q(v_1), q(v_2)) & \text{if } v_1 \in V_R \wedge v_2 \in V_R \\ 1, & \text{otherwise} \end{cases}$$

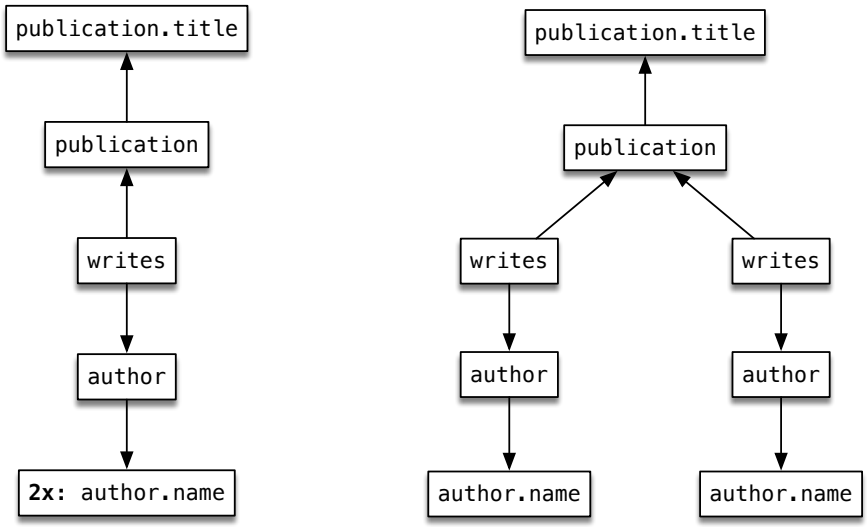
This query log-based weight function w_L returns a lower value for join edges that frequently occur in the query log.

2.5.2 Scoring Join Paths

The final score for any join path j we return is derived from the weights of the edges within the join path:

$$Score_j(j) = \frac{1}{|E_j|^2} \sum_{(v_1, v_2) \in E_j} w(v_1, v_2)$$

We divide by $|E_j|^2$ to normalize the score in a [0,1] range and also to prefer simpler join paths over more complex ones. This is based on the observations regarding *semantic relevance* [8, 43] that the closer two relations are in the schema, the likelier it is that they are semantically related.



(a) Before fork.

(b) After fork.

Figure 2.4: A simplified overview of a schema graph fork for self-joins.

2.5.3 Self-Joins

A challenge arises during join path inference when an attribute is included multiple times in the bag B_D. We present a novel approach to handling such situations to still produce valid results from the Steiner tree algorithm.

Due to the peculiarities of SQL, these situations require that our resulting join path include multiple instances of the same relation, resulting in a *self-join*. For example:

Example 2.7. *In an NLQ for the academic database, "Find papers written by both John and Jane", "John" and "Jane" both refer to attribute* `author.name`*. The correct SQL output for this NLQ is:*

```
SELECT p.title
FROM author a1, author a2, publication p,
  writes w1, writes w2
WHERE a1.name = 'John' AND a2.name = 'Jane'
  AND a1.aid = w1.aid AND a2.aid = w2.aid
```

Algorithm 4 Forking Schema Graph for Self-Joins

```
1:  function FORK(G_s, v)
2:      stack_old ← new Stack()
3:      stack_new ← new Stack()
4:      stack_old.push(v)
5:      stack_new.push(G_s.clone(v))
6:      visited ← {}
7:      while stack_old ≠ ∅ do
8:          v_old ← stack_old.pop()
9:          v_new ← stack_new.pop()
10:         visited ← visited ∪ v_old
11:         for all v_conn connected to v_old do
12:             if v_conn ∈ visited then continue
13:             if (v_old, v_conn) ∈ E_⋈ of G_s then
14:                 add edge (v_new, v_conn) to G_s
15:             else
16:                 v_cloned ← G_s.clone(v_conn)
17:                 dir ← direction of (v_old, v_conn)          ▷ ← or →
18:                 add edge (v_new, v_cloned, dir) to G_s
19:                 stack_old.push(v_conn)
20:                 stack_new.push(v_cloned)
```

```
AND p.pid = w1.pid AND p.pid = w2.pid
```

For these situations, we "fork" the schema graph, as shown (with some attribute vertices and edges removed for simplicity) in Figure 2.4, in order to account for the necessary vertices for a join path containing a self-join.

Algorithm 4 describes the process of forking the schema graph G_s in more detail, given an attribute vertex v that has been referenced multiple times. Two mirrored stacks v_{old} and v_{new} are used to track progress for the original graph and the new fork of the graph, respectively. We first clone the attribute vertex v and add it to G_s (Line 5). We repeatedly pop the top of each stack, and find all vertices v_{conn} that are connected to the current existing vertex v_{old}. We clone each v_{conn} and the edge connecting it to v_{old}, then add both to the schema graph and continue traversal (Lines 16-20). We terminate the forking process when we reach a FK-PK join edge in the direction *from* v_{old} *to* v_{conn} (Line 13). For d duplicate references to an attribute vertex v, FORK is executed $(d-1)$ times to create a

fork for each duplicate reference.

2.6 Evaluation

We performed an experimental evaluation of our system, Templar, to test whether we can use the SQL log to improve the accuracy of NLQ to SQL translation.

2.6.1 Experimental Setting

2.6.1.1 Machine Specifications

All our evaluations were performed on a computer with an 3.1 GHz Intel Core i7 processor and 16 GB RAM, running Mac OS Sierra.

2.6.1.2 Compared Systems

We enhanced two different NLIDB systems, NaLIR [43] and Pipeline, with Templar, and executed them on our benchmarks. The augmented versions are denoted NaLIR+ and Pipeline+ respectively.

The first system we augmented is NaLIR [43], a state-of-the-art pipeline-based NLIDB. We evaluated the system in its non-interactive setting because its application of user interaction is orthogonal to our approach.

We contacted authors of a few other existing NLIDBs but were not granted access to their systems. As a result, we built an NLIDB named Pipeline, which is an implementation of the keyword mapping and join path inference steps from the state-of-the-art approach in [77], excluding the hand-written repair rules. Pipeline was implemented using word2vec [51] for keyword mapping, with the default Google News corpus for calculating word similarity. While the default similarity value produced from word2vec is a cosine similarity value in the range [-1, 1], Pipeline normalizes these values to fall in the range [0, 1]. Pipeline also always selects the minimum-length join paths for join path inference.

Dataset	Size	Rels	Attrs	FK-PK	Queries
MAS	3.2 GB	17	53	19	194
Yelp	2.0 GB	7	38	7	127
IMDB	1.3 GB	16	65	20	128

Table 2.2: Statistics of each benchmark dataset.

Our implementation of Pipeline was written in Java. We used MySQL Server 5.7.18 as our relational database.

2.6.1.3 Assumptions

We assume Templar is applied in a setting where *queries in the SQL query log are representative of the SQL queries issued by users via natural language.* While this assumption does not hold true for all databases, we believe Templar is applicable for databases which already implement user-friendly interfaces such as forms or keyword search where the pattern of users' information need is likely to be similar to that of natural language interfaces.

2.6.1.4 Dataset

We tested each system by evaluating its ability to translate NLQs accurately to SQL on three benchmarks: the Microsoft Academic Search (MAS) database used in [43], and two additional databases from [77] regarding business reviews from Yelp and movie information from IMDB. Table 2.2 provides some statistics on each of these benchmark datasets.

We manually wrote the correct SQL translation for each NLQ because the original benchmarks did not include the translated SQL queries. We removed 2 queries from MAS, 1 query from Yelp, and 3 queries from IMDB because they were overly complex (i.e. contained correlated nested subqueries) or ambiguous, even for a SQL expert.

We used a cross-validation method to ensure that the test queries were not part of the SQL query log used to perform the NLQ to SQL translation. Specifically, we randomly

Dataset	System	KW (%)	FQ (%)
MAS	NaLIR	43.3	33.0
	NaLIR+	45.4	40.2
	Pipeline	39.7	32.0
	Pipeline+	**77.8**	**76.3**
Yelp	NaLIR	52.8	47.2
	NaLIR+	59.8	52.8
	Pipeline	56.7	54.3
	Pipeline+	**85.0**	**85.0**
IMDB	NaLIR	40.6	38.3
	NaLIR+	57.8	50.0
	Pipeline	32.0	27.3
	Pipeline+	**67.2**	**64.8**

Table 2.3: Keyword mapping (KW) and full query (FQ) results.

Dataset	LogJoin	FQ (%)
MAS	N	68.6
	Y	**76.3**
Yelp	N	68.5
	Y	**85.0**
IMDB	N	60.9
	Y	**64.8**

Table 2.4: Improvement from activating log-based joins in Pipeline+.

split the full dataset into 4 equally-sized folds, and performed 4 trials (one for each fold), where in each trial, the training set is comprised of 3 of the folds and the test set was the remaining fold held out of the training process. Our displayed results for all experiments are aggregated from the 4 trials.

For Pipeline and Pipeline+, we hand-parsed each NLQ into keywords and metadata to avoid any parser-related performance issues outside the scope of our work, while we passed the whole NLQ as input to NaLIR and NaLIR+ to make use of the authors' original system. For fairer comparison, we rewrote some NLQs with *wh-words* such as *who*, *what*, etc. to enable NaLIR/NaLIR+'s parser to process them correctly.

2.6.1.5 Evaluation Metrics

We measured accuracy by checking the top-ranked SQL query returned by each system by hand. For Pipeline and Pipeline+, since it was possible to return multiple queries tied for the top spot, we considered the resulting queries incorrect if there were any tie for first place.

2.6.2 Effectiveness of Templar Augmentation

In Table 2.3, we present the overall performance of each system. Pipeline+ and NaLIR+ were both executed with obscurity *NoConstOp*, $\kappa = 5$, and $\lambda = 0.8$. While all obscurity levels, including *Full* and *NoConst*, consistently improved on the baseline systems, we only show results for the best-performing obscurity level *NoConstOp* for space reasons.

2.6.2.1 Full Query

The full query (FQ) was considered correct if the NLIDB ultimately produced the correct SQL query. Pipeline+ achieves 76.3% accuracy on MAS, 85.0% accuracy on Yelp, and 64.8% accuracy on IMDB. Compared to the vanilla Pipeline system, this was a 138%, 57%, and 137% increase in accuracy, respectively. NaLIR+ improved on NaLIR by more modest margins, with a 22% increase for MAS, 12% for Yelp, and 31% for IMDB.

2.6.2.2 Keyword Mapping

For keyword mapping (KW), we considered the mapping correct if and only if all non-relation keywords were mapped correctly by the system. Pipeline's performance improved with Templar most notably for KW, with a 96%, 50%, and 110% increase for MAS, Yelp, and IMDB respectively. The improvement on NaLIR was 5% for MAS, 13% for Yelp, and 42% for IMDB.

2.6.2.3 Join Path Inference

In Table 2.4, we investigate the effect of the Join Path Generator. We focus on Pipeline+ for space reasons, and because improvement was not as drastically evident in NaLIR for reasons described in Section 2.6.3.

Activating the Join Path Generator (LogJoin "Y") increased accuracy by 11% for MAS, 24% for Yelp, and 6% for IMDB. The combined effect of this with the Keyword Mapper enabled the overall improvement through Templar.

2.6.3 Error Analysis

Augmenting Pipeline with Templar had a more dramatic effect than with NaLIR because it was given perfectly parsed keywords and metadata as input. Pipeline consequently had a much higher ceiling for improvement compared to NaLIR. While NaLIR is designed to be able to return the relevant metadata, in practice, the system's parser had trouble digesting the correct metadata from NLQs with explicit relation references, such as the token *papers* in *Return the authors who have papers in Conference X* for MAS, or other NLQs which resulted in nested subqueries. Our takeaway from this is that NLIDBs with better parsers will reap greater benefits from Templar, and are hopeful as off-the-shelf parsers have drastically improved since NaLIR's original release.

2.6.4 Impact of Parameters

In addition to the system options, there are two parameters that are required to be set in Templar: κ and λ. κ is the number of top candidate keyword mappings to return before generating configurations, and λ is the weight given to the word similarity score as opposed to the log-driven score. We observed the effects of these parameters on Pipeline+.

Figure 2.5 shows that any $\kappa \geq 5$ yields more or less consistent performance. Consequently, we chose $\kappa = 5$ as a cutoff for all our benchmarks because it reflected optimal

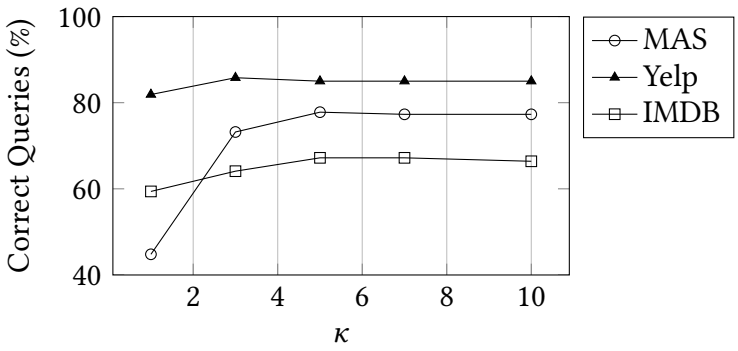


Figure 2.5: Accuracy of Pipeline+ on each benchmark given a value of κ, with λ fixed at 0.8.

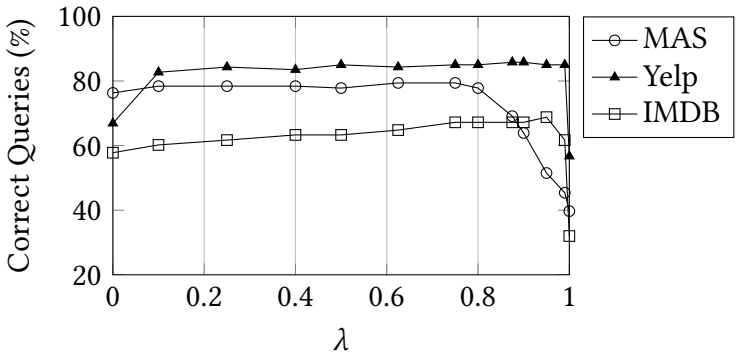


Figure 2.6: Accuracy of Pipeline+ on each benchmark given a value of λ, with κ fixed at 5.

performance and queries were also evaluated in a timely manner.

In addition, we evaluated the end-to-end performance of Pipeline+ with varying values of λ and find similar performance across all benchmarks for $0.1 \leq \lambda \leq 0.8$. For the Yelp benchmark, accuracy falls when λ is 0 because the word similarity scores are necessary when ranking configurations, while for the other benchmarks, the pruning procedure for candidate mappings is sufficient to retain and distinguish the correct mappings. Accuracy gradually drops on the MAS and IMDB benchmarks for $\lambda > 0.8$, and sharply on all benchmarks as λ approaches 1, suggesting that the log information is crucial for most queries.

2.7 Summary

In this chapter, we have described Templar, a system that enhances the performance of existing NLIDBs using SQL query logs. We model the information in the SQL query log in a data structure called the Query Fragment Graph, and use this information to improve the ability of existing NLIDBs to perform keyword mapping and join path inference. We demonstrated a significant improvement in accuracy when augmenting existing pipeline NLIDBs using log information with Templar. Possible future work includes exploring the influence of user sessions in the SQL query log, as well as finding ways to improve existing deep learning-based end-to-end NLIDBs with information from the SQL log.

CHAPTER 3

Combining Natural Language and Programming-by-Example[1]

3.1 Introduction

As mentioned in Chapter 1, 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 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 [57,61,77] or are overly reliant on specific sentence structures [43]. More recent advances leverage deep learning in an attempt to circumvent these challenges, but the state-of-the-art accuracy [79] 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 [57].

Another alternative to writing SQL is programming-by-example (PBE), where users

[1]©2020 ACM. This is the author's version of the work. The definitive Version of Record was published in: Christopher Baik, Zhongjun Jin, Michael Cafarella, and H. V. Jagadish, *Duoquest: A Dual-Specification System for Expressive SQL Queries*, Proceedings of the 2020 ACM International Conference on Management of Data, (SIGMOD '20), June 14–19, 2020, Portland, OR, USA, http://dx.doi.org/10.1145/3318464.3389776.

System	Soundness	Query Expr.[2]			Knowledge[3]		
		⋈	σ	γ	NS	PT	OW
NLIs [43, 77, 79]		✓	✓	✓	✓	N/A	N/A
PBE Systems							
QBE [82]	✓	✓	✓	✓		✓	✓
MWeaver [59]	✓	✓			✓		✓
S4 [58]	✓	✓			✓	✓	✓
SQuID [25]	✓	✓	✓	✓[4]	✓		✓
TALOS [69]	✓	✓	✓	✓	✓		
QFE [44]	✓	✓	✓				
PALEO [55]	✓		✓	✓			
Scythe [73]	✓	✓	✓	✓			
REGAL+ [65]	✓	✓	✓	✓	✓		
DUOQUEST	✓	✓	✓	✓	✓	✓	✓

Table 3.1: DUOQUEST vs. NLI/PBE, considering soundness, query expressiveness, and required user knowledge. A ✓ is desirable in each column.

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

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 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 type of 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.

Second, we want to *minimize user time 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:

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 previous single-specification approaches.

2. We efficiently explore the search space of candidate queries with *guided partial query enumeration* (GPQE), 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 describe our implementation of DUOQUEST, a *novel prototype dual-specification system*, which leverages GPQE and a front-end web interface with autocomplete functionality for literal values.

3. We present user studies on DUOQUEST demonstrating that the dual-specification approach enables a 62.5% absolute increase in accuracy over a state-of-the-art NLI and comparable accuracy to a PBE system on a more limited workload for the PBE system. We also present a simulation study on the Spider benchmark demonstrating a >2x increase in the top-1 accuracy of DUOQUEST over both NLI and PBE.

Organization — In Section 3.2, we provide an overview of our problem. We then describe our solution approach (Section 3.3) and system implementation (Section 3.4). We present our experimental evaluation, including user studies and simulated experiments

(Section 3.5), explore related work (Section 3.6), discuss limitations of our approach and opportunities for future work (Section 3.7), and summarize (Section 3.8).

3.2 Problem Overview

3.2.1 Motivating Example

Consider the following motivating example:

Example 3.1. *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 3.1 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 3.1 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:

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

Table 3.2: 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.

Example 3.2. *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 3.2.

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.

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.

3.2.2 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 [58, 59]. Unlike existing approaches, we also allow the user to include some additional metadata about their desired SQL query:

Definition 3.1 (R6:D3)**.** *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*[5]*.*

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 corresponding cell of the same index $\chi_i[j] \in \chi_i$. As shown in Example 3.2, 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 3.2. *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;*

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

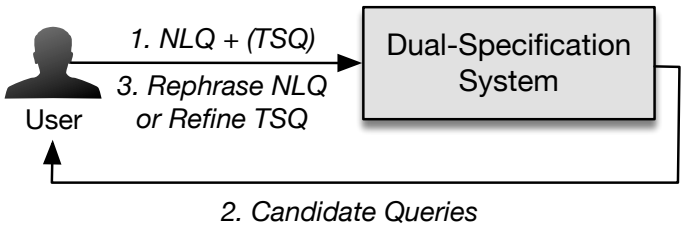


Figure 3.1: Dual-specification interaction model.

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.

3.2.3 Problem Definition

We now formally define our dual-specification problem:

Problem 3.1. *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 3.4.

3.2.4 Interaction

Figure 3.1 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.

3.2.5 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.

3.3 Solution Approach

3.3.1 Overview

The search space of possible SQL queries in our setting is enormous[6], 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 [75]. The set of queries we hope to support only further expands this search space.

Previous work [74] 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.

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

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 3.3. *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 [77] or can be easily adapted [74] 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 each one. Ultimately, this enables the approach to cover more of the search space in a given amount of time.

3.3.2 Algorithm

Algorithm 5 describes the GPQE 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

Algorithm 5 Guided Partial Query Enumeration

```
1:  function ENUMERATE(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
```

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 3.2 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.

3.3.3 Guided Enumeration

In this section, we describe the enumeration process in EnumNextStep. We adopt the SyntaxSQLNet [79] 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 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.

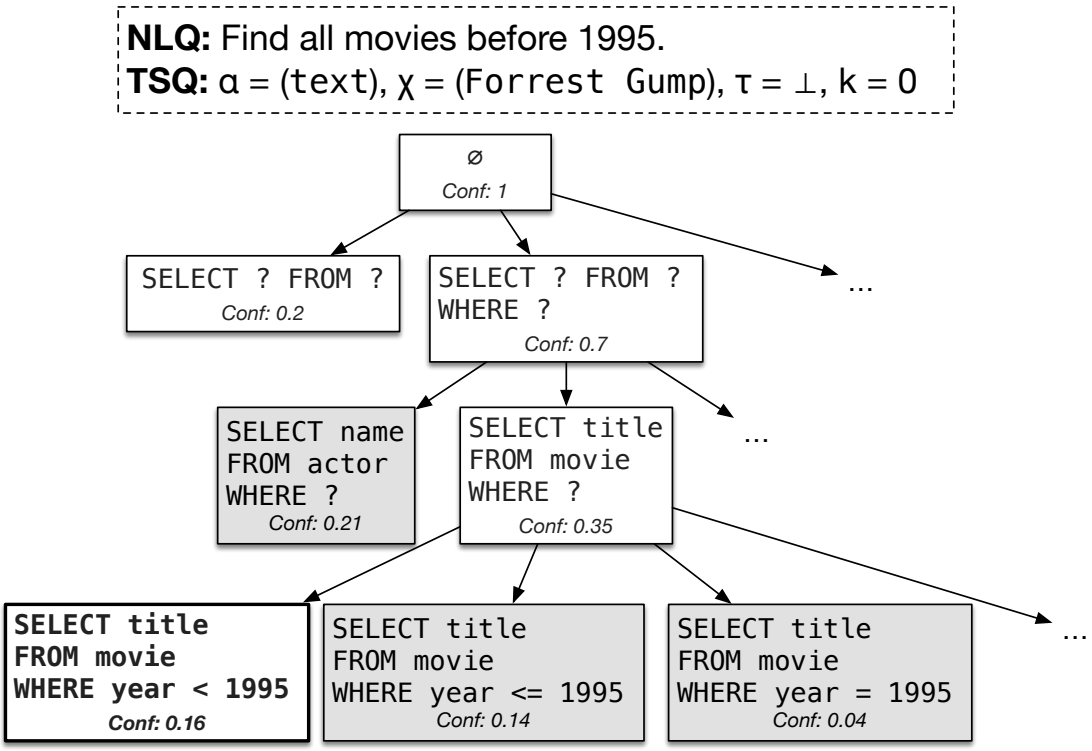


Figure 3.2: Simplified GPQE example. Each box is a state. Shaded boxes fail verification against the TSQ. The bolded state is the highest-ranked candidate query.

3.3.3.1 Background

SyntaxSQLNet uses a collection of recursive neural network modules, each responsible for making an enumeration decision for a specific SQL syntax element. We list the modules used in our system in Table 3.3. Each module takes the natural language query N, the partial query synthesized so far p, and optionally, the database schema D (for modules such as the `COL` module which infer a column from the database schema). 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.

The order of module execution is pre-assigned based on SQL syntax rules and the current output state p. For example, if a `WHERE` clause is being predicted, the `COL`, `OP`, and

Module	Responsibility	Output
KW	Clauses present in query (WHERE, GROUP BY, ORDER BY)	Set
COL	Schema columns	Set
OP	Predicate operators (e.g. =, LIKE)	Set
AGG	Aggregate functions (MAX, MIN, SUM, COUNT, AVG, None)	Set
AND/OR	Logical operators for predicates	Single
DESC/ASC	ORDER BY direction and LIMIT	Single
HAVING	Presence of HAVING clause	Single

Table 3.3: Selected modules from SyntaxSQLNet [79], their respective responsibility and output cardinality.

ROOT/TERM modules will be executed in order.

3.3.3.2 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.

3.3.3.3 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 5, 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:

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

Algorithm 6 Progressive Join Path Construction

```
1:  function ConstructJoinPaths(q, D)
2:      C ← get all column references in q
3:      T ← get all tables encompassing C
4:      R ← ∅
5:      if |T| = 0 then
6:          R ← tables in D
7:      else
8:          J ← Steiner(T, D)
9:          add J to R
10:         for t ∈ FKs to PKs in T do
11:             J′ ← AddJoin(J, t)
12:             add J′ to R
13:     return R
```

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 3.1. *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 [77] 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.

3.3.3.4 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 [5]. 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 a technique called *progressive join path construction*. Algorithm 6 describes the join path construction process, which takes q, a partial query, and D, the database as input. First, the set of distinct tables encompassing all column references in q are collected into T (Line 3). If there are no tables present in the query (e.g. `SELECT COUNT(*)`), then each table in D is returned as a candidate join path (Line 6). Otherwise, following the approach in [5], a Steiner tree is computed on the graph where nodes are tables and edges are foreign key to primary key relationships between the tables (Line 8). By default, all edge weights are set to 1, though weights could also be derived from sources such as a query log [5]. Finally, in Lines 10-12, we add joins to cover cases where the desired query contains additional tables in the `FROM` clause beyond the columns already present in q, such as in the following example.

Example 3.3. *A query utilizing more tables than those referenced outside the* `FROM` *clause:*

```
SELECT a.name FROM actor a
  JOIN starring s ON a.aid = s.aid
```

The process in Lines 10-12 can be recursively called to add joins of arbitrary depth. For simplicity, we only depict the process for one level of depth in Algorithm 6.

Whenever a new partial query is generated, progressive join path construction is executed to produce a new state for each candidate join path of the partial query. While all

states produced by this process have the same confidence score, the enumeration process prioritizes states with higher confidence scores first, and then uses the join path length as a secondary tiebreaker, where shorter join paths are preferred.

3.3.3.5 Extensibility

As NLI models are undergoing rapid active development in the programming languages [77], natural language processing [11, 30, 79], and database research communities [43], 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 3.1.

3.3.3.6 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 3.2.5.

3.3.4 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 5. Low-cost verifications, which do not require any access to the database D, are performed first to avoid performing high-cost verifications,

Algorithm 7 Verification

```
1:  function VERIFY(𝒯, L, q, D)
2:      α, χ, τ, k = 𝒯
3:      if ¬VERIFYCLAUSES(τ, k, q) then return ⊥
4:      if ¬VERIFYSEMANTICS(q) then return ⊥
5:      if ¬VERIFYCOLUMNTYPES(α, q, D) then return ⊥
6:      if ¬VERIFYBYCOLUMN(χ, q, D) then return ⊥
7:      if CANCHECKROWS(q) then
8:          if ¬VERIFYBYROW(χ, q, D) then return ⊥
9:      if q is complete then
10:         if ¬VERIFYLITERALS(q, L) then return ⊥
11:         if τ ∧ |χ| >= 2 then
12:             if ¬VERIFYBYORDER(χ, q, D) then return ⊥
13:     return ⊤
```

which involve issuing queries on D, until absolutely 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. Algorithm 7 describes this process, which takes the TSQ $\mathcal{T}$, a partial query q, the literal values L within the natural language query, and the database D as input.

First, the presence of clauses is verified in VERIFYCLAUSES. If the TSQ specifies that results should be sorted or limited and the partial query does not match the TSQ, verification will fail. For example:

Example 3.4. *Given a TSQ with sorting flag $\tau = \bot$ and the following partial queries, where* ? *indicates a placeholder:*

CQ1: `SELECT name, birth_yr FROM actor WHERE ?`

CQ2: `SELECT name, birthplace FROM actor WHERE ?`

CQ3:
```
SELECT a.name, COUNT(*) FROM actor a JOIN
    starring s ON a.aid = s.aid GROUP BY a.name
```

CQ4:
```
SELECT a.name, MAX(m.revenue) FROM actor a
  JOIN starring s ON a.aid = s.aid JOIN
```

Error	Description	Example	Possible Alternative
Inconsistent predicates	Do not permit selection predicates on the same column that contradict each other.	`SELECT name FROM actor WHERE name = 'Tom Hanks' AND name = 'Brad Pitt'`	`SELECT name FROM actor WHERE name = 'Tom Hanks' OR name = 'Brad Pitt'`
Constant output column	Do not permit columns with equality predicates to be projected.	`SELECT name, birth_yr FROM actor WHERE birth_yr = 1950`	`SELECT name FROM actor WHERE birth_yr = 1950`
Ungrouped aggregation	An unaggregated projection and aggregation cannot be used together without `GROUP BY`.	`SELECT birth_yr, COUNT(*) FROM actor`	`SELECT birth_yr, COUNT(*) FROM actor GROUP BY birth_yr`
`GROUP BY` w/singleton groups	If each group consists of a single row (e.g. group contains primary key), aggregation is unneeded.	`SELECT aid, MAX(birth_yr) FROM actor GROUP BY aid`	`SELECT aid, birth_yr FROM actor`
Unnecessary `GROUP BY`	If there are no aggregates in the `SELECT`, `ORDER BY` or `HAVING` clauses, `GROUP BY` is unnecessary.	`SELECT name FROM actor GROUP BY name`	`SELECT name FROM actor`
Aggregate type usage	`MIN/MAX/AVG/SUM` may not be applied to text columns.	`SELECT AVG(name) FROM actor`	N/A
Faulty type comparison	`>, <, >=, <=, BETWEEN` may not be applied to text columns.	`SELECT name FROM actor WHERE name >= 'Tom Hanks'`	N/A
	`LIKE` may not be applied to numeric columns.	`SELECT birth_yr FROM actor WHERE birth_yr LIKE '%1956%'`	N/A

Table 3.4: List of semantic pruning rules. Rules may be modified depending on the domain and use case.

```
movies m ON m.mid = s.mid GROUP BY a.name
```

CQ5: `SELECT name, debut_yr FROM actor ORDER BY ?`

CQ5 would fail VerifyClauses *because the TSQ specifies that results are not to be ordered in the desired query, yet it contains an* `ORDER BY` *clause.*

Second, semantic checks are performed on the query in VerifySemantics. This step constrains the search space by eliminating nonsensical or redundant yet syntactically-correct SQL queries. Over 40 such errors are cataloged in [15]. We check for a subset of these errors and some additional ones, listed in Table 3.4. While expert users may opt

to intentionally write SQL queries that break some of these rules, we enforce these rules to constrain the set of produced queries to those even non-technical users can readily understand.

Third, the column types in the SELECT clause are verified against the types in the TSQ in VerifyColumnTypes, which requires a check on the schema of D, but still without any need to query D:

Example 3.5. *Of the remaining queries CQ1-CQ4 in Example 3.4, given a TSQ with type annotations* α = [text, number]*, CQ2 would fail VerifyColumnTypes because the types of its projected columns in the* SELECT *clause are* [text, text].

Fourth, in VerifyByColumn, tuples in the TSQ are compared column-wise against the SELECT clause of each partial query. This requires running relatively inexpensive *column-wise verification queries* on the database D:

Example 3.6. *Given an example tuple in the TSQ* χ_1 = [Tom Hanks, [1950, 1960]] *and the queries CQ1, CQ3, and CQ4 from Example 3.4, VerifyByColumn executes the following column-wise verification queries on the database:*

CV1:
```
SELECT 1 FROM actor
 WHERE name = 'Tom Hanks' LIMIT 1
```
(for 1st projected column of CQ1, CQ3, and CQ4)

CV2:
```
SELECT 1 FROM actor WHERE birth_yr >= 1950
 AND birth_yr <= 1960 LIMIT 1
```
(for 2nd projected column of CQ1)

CV3:
```
SELECT 1 FROM movies WHERE revenue >= 1950
 AND revenue <= 1960 LIMIT 1
```
(for 2nd projected column of CQ4)

CV3 is the only one producing an empty result set on D, thus causing CQ4 to fail VerifyByColumn.

For column-wise verification queries, `SELECT 1` and `LIMIT 1` are used to minimize the execution time on typical SQL engines. Each unaggregated projected column in the `SELECT` clause of the partial query is matched against the corresponding cell in the example tuple, whether via an equality operator for single-valued cells in the tuple or `>=`/`<=` operators for range cells, and placed in the `WHERE` clause, while the `FROM` clause is assigned as the table of the projected column. Aggregated projections with `MIN` or `MAX` are treated the same as unaggregated projections, as both these functions will produce an exact value from the projected column. For `AVG`, the range (i.e. minimum value to maximum value) of the projected column is compared with the range cell, and verification fails if the two ranges do not intersect. Projections with `COUNT` and `SUM` aggregations are ignored because no conclusion can easily be drawn for partial queries.

Fifth, row-wise verification is performed. CanCheckRows enforces the precondition for row-wise verification: any partial query with aggregated projections needs completed `WHERE`/`GROUP BY` clauses with no holes, because completing those holes could change the output of the aggregated projections in the final query. *Row-wise verification queries* are similar to column-wise verification queries, except that they require output values of each partial query to reside in the same tuple when matched with example tuples in the TSQ:

Example 3.7. *Given the example tuple χ_1 from Example 3.6 and the queries CQ1 and CQ3 from Example 3.4,* VerifyByRow *executes the following row-wise verification queries on the database for CQ1 and CQ3 respectively:*

RV1:
```
SELECT 1 FROM actor WHERE name = 'Tom Hanks'
    AND (birth_yr >= 1950 AND birth_yr <= 1960)
    LIMIT 1
```

RV2:
```
SELECT 1 FROM actor a JOIN starring s ON
```

```
a.aid = s.aid WHERE name = 'Tom Hanks'
GROUP BY a.name HAVING (COUNT(*) >= 1950 AND
COUNT(*) <= 1960) LIMIT 1
```

RV1 produces a valid result on D, while RV2 does not. As a result, CQ1 is the only CQ that passes all verification tests.

Each projected column in the SELECT clause of the candidate query is matched against the corresponding cell in the example tuple and appended to either the WHERE (for unaggregated projections) or HAVING (for aggregated projections) of the column-wise verification query. All other elements from the original candidate query (such as FROM, GROUP BY clauses, or other selection predicates) are retained in the row-wise verification query.

Finally, when the query q is complete, the algorithm verifies that all literals L are used in q via VerifyLiterals. Then, if multiple example tuples exist in the TSQ and the sorting flag $\tau = \top$, VerifyByOrder executes q on D and ensures that each of the example tuples in χ is fulfilled in the same order as they were specified in the TSQ.

3.3.5 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.

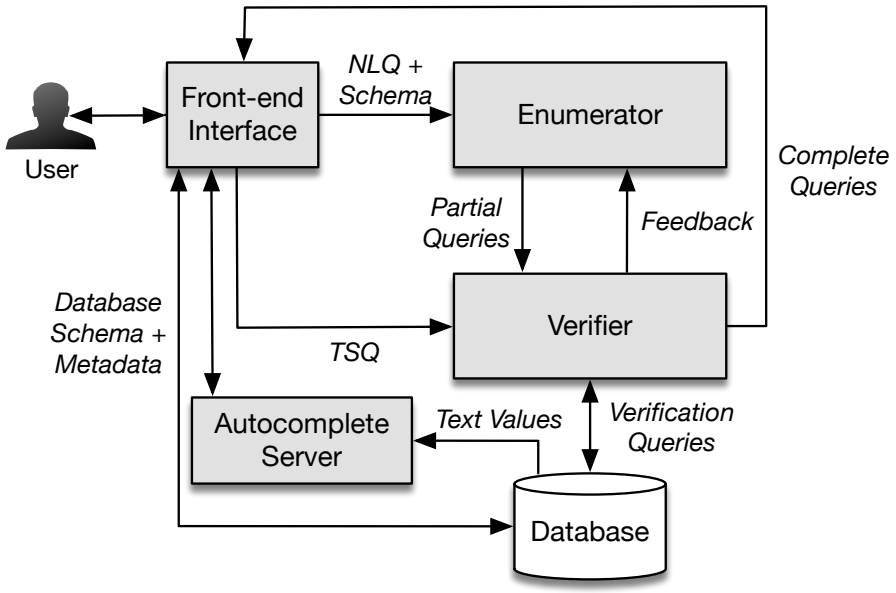


Figure 3.3: Architecture of DUOQUEST.

3.4 Implementation

We implemented our approach in a prototype system, DUOQUEST. The system architecture (Figure 3.3) is comprised of 4 micro-services: the Enumerator, Verifier, Front-end Interface, and Autocomplete Server.

The Enumerator performs the ENUMNEXTSTEP procedure, and uses a SyntaxSQL-Net [79] model pre-trained using the training and development sets of the cross-domain Spider dataset [80], while the Verifier service executes VERIFY.

The Front-End Interface (Figure 3.4) enables the user to specify queries. The interface contains a search bar for the user to specify the NLQ. Users can specify domain-specific literal text values in the NLQ search bar by typing the double-quote (") character, which activates an autocomplete search over a master inverted column index [63] containing all text columns in the database. The TSQ interface is below the search bar, where each cell in the interface activates the same autocomplete search as literal text values are typed.

After issuing the query, candidate SQL queries are displayed one at a time from highest to lowest confidence as the system enumerates and verifies them. Candidate queries

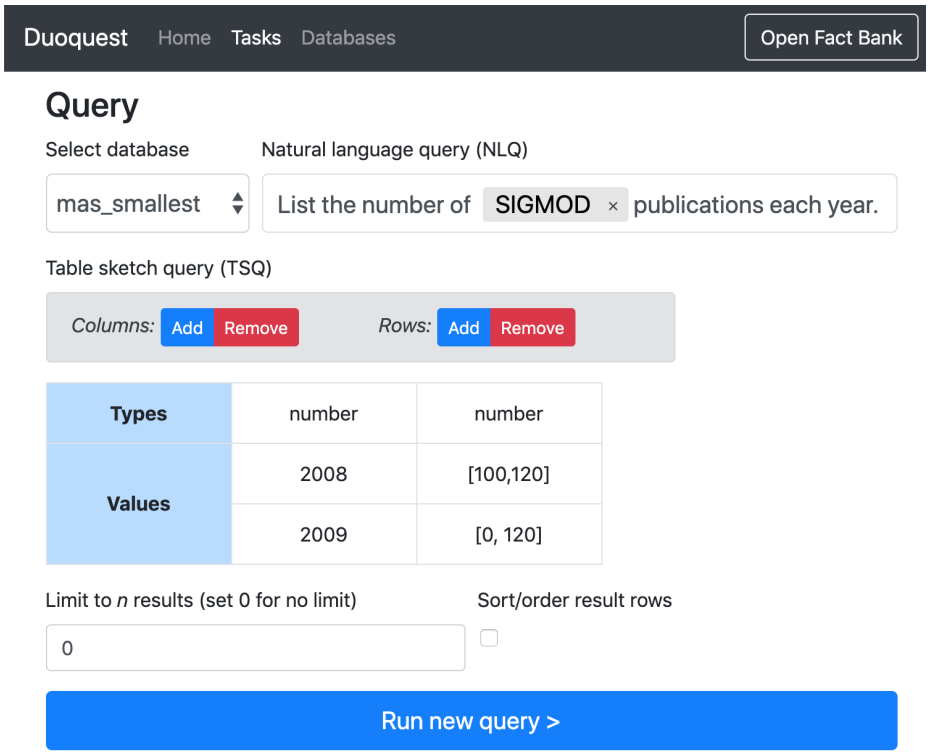


Figure 3.4: Screenshot of front-end interface. The "SIGMOD" tag was produced via auto-complete.

continue to load until a pre-specified timeout is exceeded or the user clicks the "Stop Task" button. To enable users without knowledge of SQL to distinguish candidate queries and select from among them, each candidate query has a "Query Preview" button which executes the query on the database with `LIMIT 20` appended to the query to retrieve a 20-row preview of the query results, and a "Full Query View" which executes the full query on the database.

3.4.1 Domain-Specific Customization

Adapting DUOQUEST to a new domain requires minimal effort, as the NLI model is trained on a cross-domain corpus. Additional domain-specific tasks can be used to retrain the model, and domain-specific semantic rules may also be appended to the default semantic rules provided by DUOQUEST. New databases should have foreign key-primary key con-

straints explicitly defined on the schema for the system to ingest (or these can be manually specified on our administrator's interface), and table and column names should use complete words rather than abbreviations (e.g. `author_id` instead of `aid`) as the NLI model relies on off-the-shelf word embedding models to interpret NLQs.

3.5 Evaluation

We explored several research questions in our evaluation:

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 time over single-specification approaches?

RQ3: How does each component of our algorithm contribute to system performance?

RQ4: How does the amount of detail provided in the TSQ affect system performance?

3.5.1 Setup for User Studies

3.5.1.1 Compared Systems

For **RQ1**/**RQ2**, we conducted two within-subject user studies: one between DUOQUEST and SyntaxSQLNet [79], a state-of-the-art NLI; and the other with DUOQUEST and SQuID [25], a state-of-the-art PBE system.

We selected SyntaxSQLNet as a representative end-to-end neural network NLI. While some recent NLIs [11, 30] 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.

Experiment	Dataset	# DBs	Tasks				Avg. Schema Stats		
			E	M	H	Total	Tbls	Cols	FK-PK
User Study vs. NLI	MAS [43]	1	0	3	5	8	15	44	19
User Study vs. PBE	MAS [43]	1	0	4	2	6	15	44	19
Simulation	Spider Dev [80]	20	239	252	98	589	4.1	22.1	3.2
	Spider Test [80]	40	524	481	242	1247	4.5	19.6	3.6

Table 3.5: 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* (E) tasks were project-join queries including aggregates, sorting, and limit operators, *Medium* (M) tasks also included selection predicates, and *Hard* (H) tasks included grouping operators.

We selected SQuID as the representative PBE system because, to the best of our knowledge (Table 3.1), 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.

3.5.1.2 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.

3.5.1.3 Tasks

We tested DUOQUEST against NLI on a variety of tasks within the scope described in Section 3.2.5. 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 3.5) to see if they could synthesize the desired SQL query matching the provided task description. Each task description was provided in Chinese[8] following the study procedure in [43] to force the user to articulate the NLQ in English using their own words. 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 performing the study to teach them how to use each system.

The tasks were split into two sets per user study: A/B for the NLI study (Table 3.7) and C/D for PBE (Table 3.8). 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.

3.5.1.4 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

[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.

SQL selected queries using a combination of eyeballing the selection predicates in the SQL queries and observing the "Query Preview" (described in Section 3.4) 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.

3.5.1.5 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 control for 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.

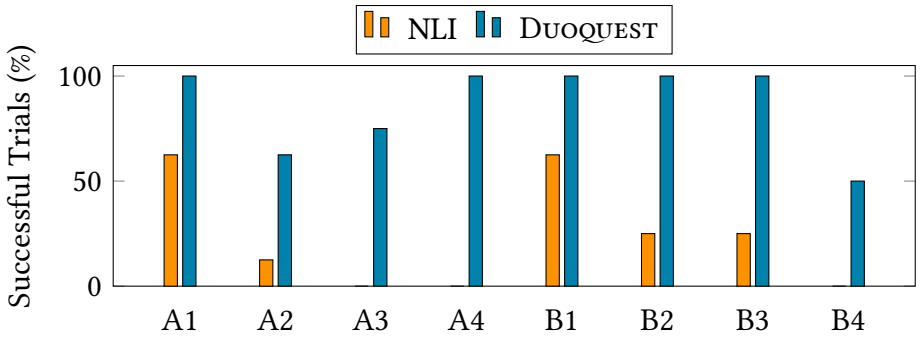


Figure 3.5: % of trials for NLI study in which the user successfully completed each task within 5 minutes.

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

3.5.1.6 Environment

For DUOQUEST and NLI, a server was set up on a Ubuntu 16.04 machine with 16 2.10 GHz Intel Xeon Gold 6130 CPUs and 4 NVIDIA GeForce GTX 1080 Ti GPUs (only a single GPU was used for inference), running PyTorch 0.4.0 on CUDA 7.5. The front end was accessed with a MacBook Pro using Google Chrome. PBE was executed on a Java graphical user interface on a MacBook Pro.

3.5.2 User Study vs. NLI

Figure 3.5 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 suc-

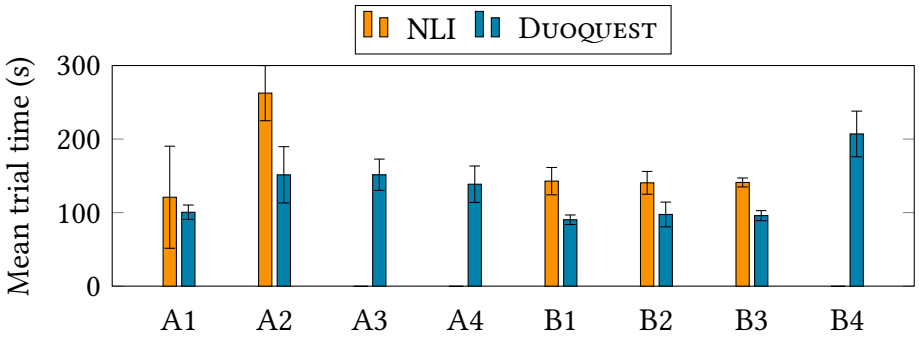


Figure 3.6: Mean time per task for correctly completed trials in NLI study, with error bars indicating standard error. A3, A4, B4 for NLI are omitted because there were no successful trials.

cessful 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 observe in Figure 3.6 that DUOQUEST ***either reduces or requires comparable user time 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.

3.5.3 User Study vs. PBE

For **RQ1**, Figure 3.7 shows that DUOQUEST ***and PBE have comparable accuracy on the PBE-supported workload***, with DUOQUEST performing marginally better on the more difficult Hard tasks (C3, D3).

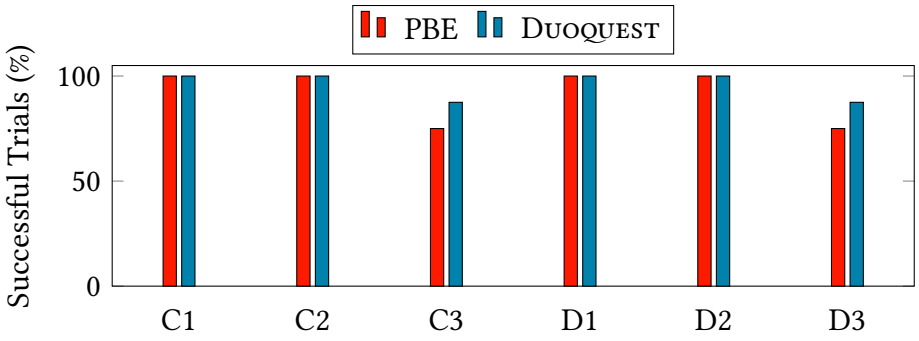


Figure 3.7: % of trials for PBE study in which the user successfully completed each task within 5 minutes.

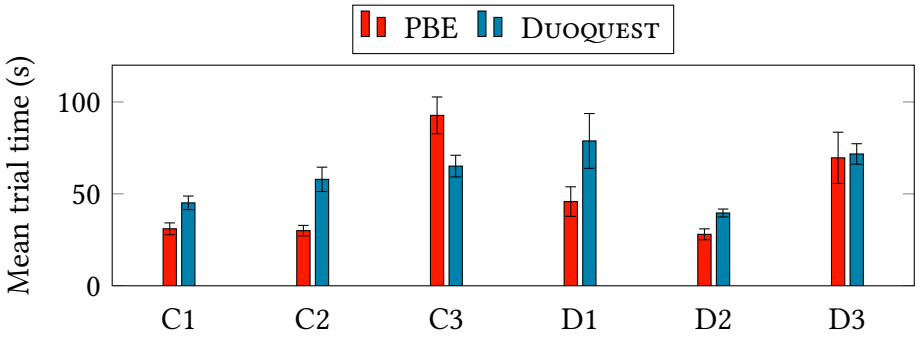


Figure 3.8: Mean time per task for correctly completed trials in PBE study; error bars for standard error.

For **RQ2**, Figure 3.8 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 3.9 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.

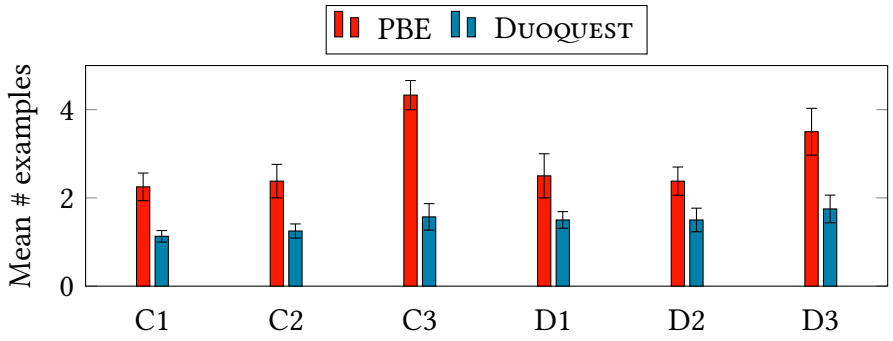


Figure 3.9: Mean # examples used per task for successful trials in PBE study; error bars for standard error.

3.5.4 Simulation Study

3.5.4.1 Setup

We evaluated DUOQUEST on the Spider benchmark [80], 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 3.2.5), 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 3.5) 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

System	Top-1		Top-10		Correct		Unsupported	
	#	%	#	%	#	%	#	%
DUOQUEST	**374**	**63.5**	**493**	**83.7**	-	-	0	0
NLI	178	30.2	334	56.7	-	-	0	0
PBE	-	-	-	-	78	13.2	475	80.6

(a) Spider Dev (589 total tasks)

System	Top-1		Top-10		Correct		Unsupported	
	#	%	#	%	#	%	#	%
DUOQUEST	**792**	**63.5**	**1065**	**85.4**	-	-	0	0
NLI	389	31.2	698	56.0	-	-	0	0
PBE	-	-	-	-	203	16.3	972	77.9

(b) Spider Test (1247 total tasks)

Figure 3.10: Top-1 and Top-10 accuracy for DUOQUEST and NLI, task correctness for PBE, and amount of unsupported tasks.

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.

3.5.4.2 Accuracy

Figure 3.10 displays the results of DUOQUEST and NLI's top-k accuracy, which is the number of tasks for which the desired query appeared in the top-k of returned candidate queries. In particular, the Top-10 accuracy is a good proxy for the user's ability to discover their desired query, as we consider that examining a list of 10 candidate queries is a reasonable burden for the user to carry.

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 and selection

	Easy			Medium			Hard		
System	✓#	✓%	U#	✓#	✓%	U#	✓#	✓%	U#
DUOQUEST	**218**	**91.2**	0	**214**	**84.9**	0	**61**	**62.2**	0
NLI	158	66.1	0	143	56.8	0	33	33.8	0
PBE	29	12.1	210	49	19.4	167	0	0	98

(a) Spider Dev (239 easy, 252 medium, 98 hard tasks)

	Easy			Medium			Hard		
System	✓#	✓%	U#	✓#	✓%	U#	✓#	✓%	U#
DUOQUEST	**495**	**94.5**	0	**407**	**84.6**	0	**163**	**67.4**	0
NLI	379	72.3	0	246	51.1	0	73	30.2	0
PBE	107	20.4	417	96	20.0	313	0	0	242

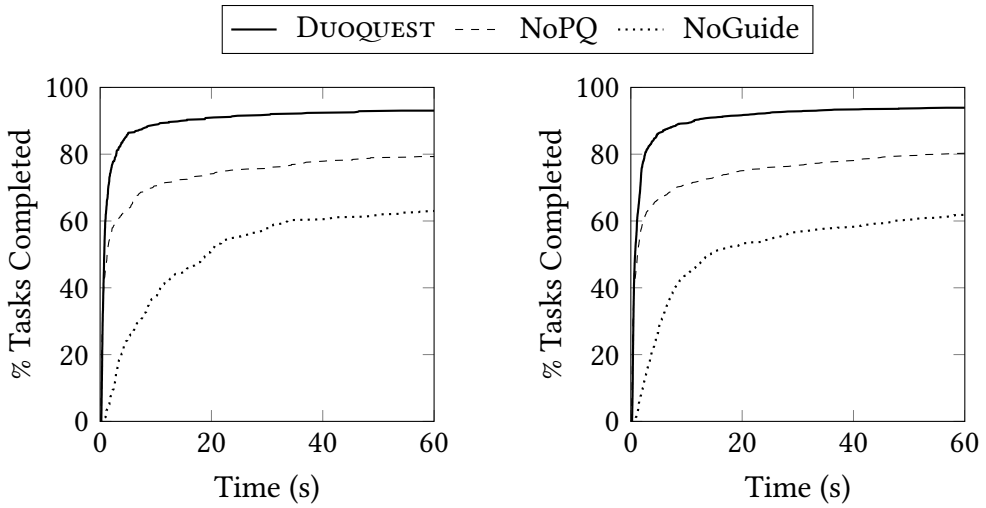
(b) Spider Test (524 easy, 481 medium, 242 hard tasks)

Figure 3.11: Number (✓#) and proportion (✓%) of correct tasks (top-10 accuracy for DUOQUEST and NLI) and number of unsupported tasks (U#) by task difficulty level.

predicates with negation or `LIKE` operators. For tasks the PBE system could support, we did not measure top-k accuracy because the expected interaction model differed from the other systems. Instead, we labeled the result *Correct* if the selection predicates in the desired query were a subset of PBE's produced candidate selection predicates, ignoring any differences in specific literal values.

Reinforcing our conclusions on **RQ1** from the user study, DUOQUEST handily beats single-specification approaches NLI and PBE, ***with a >2x increase in Top-1 accuracy and 47.6% increase in Top-10 accuracy*** over NLI, and an even larger improvement over PBE on the development set. Results are similar on the test set.

Figure 3.11 presents a breakdown of task success by difficulty level, measured by top-10 accuracy for DUOQUEST and NLI and correctness for PBE. As expected, systems perform generally worse on more difficult tasks as the resulting SQL for harder tasks contained more complex query constructs. PBE was unable to support any hard tasks because they all included projected aggregate values.



(a) Spider Dev

(b) Spider Test

Figure 3.12: Distributions of the time taken for each algorithm to synthesize the correct query. A higher curve indicates superior performance.

While PBE should have been able to get all supported tasks correct, it failed several tasks to due to its requirements for a star/snowflake schema and user-defined metadata annotations as to which schema attributes are "entities" or "concepts". While we offered our best effort in restructuring and labeling the schema so as 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.

3.5.4.3 Guided Partial Query Enumeration (GPQE)

To answer **RQ3**, we selectively disabled the two components of the GPQE algorithm used in DUOQUEST: guided enumeration (Section 3.3.3) and pruning of partial queries (Section 3.3.4). The version without guided enumeration (*NoGuide*) used only the literals from the NLQ specification and performed a naïve breadth-first search enumeration of all possible queries (ignoring confidence scores) while still pruning partial queries when possible. Simpler queries (i.e. those with less operations) were enumerated first and column attributes were enumerated following the order of the schema metadata provided in

	Spider Dev			Spider Test		
Detail	T1	T10	T100	T1	T10	T100
Full	**63.5**	**83.7**	**91.7**	**63.5**	**85.4**	**92.4**
Partial	59.6	77.1	90.3	58.6	81.5	90.5
Minimal	40.8	60.6	85.9	41.1	68.6	85.1
NLI	*30.2*	*56.7*	*69.4*	*31.2*	*56.0*	*69.5*

Table 3.6: Top-1, Top-10, and Top-100 exact matching accuracy (%) for TSQs with varying amounts of specification detail. NLI results shown for comparison.

the Spider benchmark. The algorithm disabling pruning of partial queries (*NoPQ*) leveraged enumeration guidance, but only verified complete queries, not partial ones, making it identical to the naïve chaining approach described in Section 3.3.5.

Figure 3.12 displays the results. In theory, all these systems explore the same search space, and given enough time, the distributions will all converge. In practice, however, the user cannot wait indefinitely, and the figure demonstrates how ***performance suffers immensely when we disable either guided enumeration or the pruning of partial queries***, highlighting their necessity in facilitating an efficient, interactive-time system.

3.5.4.4 Specification Detail

To answer **RQ4**, we varied the amount of detail in the synthesized TSQ provided to DuoQuest. We considered three different levels of detail:

(1) *Full*, using the full synthesized TSQ described in Section 3.5.4.1;

(2) *Partial*, for which all values for a randomly-selected single column in tasks with at least 2 projected columns were erased from example tuples in the *Full* TSQ;

(3) *Minimal*, which removes all example tuples from the TSQ, leaving only column type annotations.

Table 3.6 demonstrates how an ***increase in specification detail helps contribute to a corresponding increase in the performance of*** DuoQuest. Performance for the

Partial TSQ has a relatively small dropoff from the Full TSQ, showing the promise of using partial or incomplete tuple knowledge to help users construct queries. There is a larger gap between Partial and Minimal TSQs, suggesting that the presence of even a single partial tuple is preferable to no example tuples at all. Finally, even providing type annotations for each column allows a 30% improvement in top-1 accuracy over the baseline NLI system which uses no TSQ.

3.6 Related Work

Natural language interfaces (NLIs) Most early NLIs for relational databases were confined to a single domain [3]. 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 [57]. Other systems utilized different technologies such as dependency parse trees [43], semantic parsing [77], or pre-defined ontologies [61] 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 [79], graph neural networks [11], or an intermediate representation [30] 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 [49], a representative sample of which is displayed in Table 3.1. 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 [25]

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.

3.7 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 3.5.1.4), 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 produced candidate SQL queries against 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.

3.8 Summary

In this chapter, 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.

Task	Level	English Description	SQL
A1	M	List all publications in conference C and their year of publication.	`SELECT t2.title, t2.year FROM conference AS t1 JOIN publication AS t2 ON t1.cid = t2.cid WHERE t1.name = 'C'`
A2	H	List keywords and the number of publications containing each, ordered from most to least publications.	`SELECT t1.keyword, COUNT(*) FROM keyword AS t1 JOIN publication_keyword AS t2 ON t1.kid = t2.kid JOIN publication AS t3 ON t2.pid = t3.pid GROUP BY t1.keyword ORDER BY count(*) DESC`
A3	H	How many publications has each author from organization R published?	`SELECT t1.name, COUNT(*) FROM author AS t1 JOIN writes AS t2 ON t2.aid = t1.aid JOIN organization AS t3 ON t3.oid = t1.oid JOIN publication t4 ON t4.pid = t2.pid WHERE t3.name = 'R' GROUP BY t1.name`
A4	H	List journals with more than 500 publications and the publication count for each.	`SELECT DISTINCT t1."name", COUNT(*) FROM journal AS t1 JOIN publication AS t2 ON t1.jid = t2.jid GROUP BY t1.name HAVING COUNT(*) > 500`
B1	M	List the titles and years of publications by author A.	`SELECT t1.title, t1.year FROM publication AS t1 JOIN writes AS t2 ON t2.pid = t1.pid JOIN author AS t3 ON t3.aid = t2.aid WHERE t3.name = 'A'`
B2	M	List the conferences and homepages in the D domain.	`SELECT t1.name, t1.homepage FROM conference AS t1 JOIN domain_conference AS t2 ON t2.cid = t1.cid JOIN domain AS t3 ON t3.did = t2.did WHERE t3.name = 'D'`
B3	H	List organizations with more than 100 authors and the number of authors for each.	`SELECT t2.name, COUNT(*) FROM author AS t1 JOIN organization AS t2 ON t1.oid = t2.oid GROUP BY t2.name HAVING COUNT(*) > 100`
B4	H	List authors from organization R with more than 50 publications and the number of publications for each author.	`SELECT t1.name, COUNT(*) FROM author AS t1 JOIN writes AS t2 ON t1.aid = t2.aid JOIN organization AS t3 ON t1.oid = t3.oid JOIN publication AS t4 ON t2.pid = t4.pid WHERE t3.name = 'R' GROUP BY t1.name HAVING COUNT(*) > 50`

Table 3.7: Tasks for the user study vs. NLI, with abbreviated foreign key names and literal values.

Task	Level	English Description	SQL
C1	M	List all publications in conference C.	`SELECT t2.title FROM conference AS t1 JOIN publication AS t2 ON t1.cid = t2.cid WHERE t1.name = 'C'`
C2	M	List authors in domain D.	`SELECT t1.name FROM author AS t1 JOIN domain_author AS t2 ON t1.aid = t2.aid JOIN domain AS t3 ON t2.did = t3.did WHERE t3.name = 'D'`
C3	M	List authors with more than 5 papers in conference C.	`SELECT t1.name FROM author AS t1 JOIN writes AS t2 ON t1.aid = t2.aid JOIN publication AS t3 ON t2.pid = t3.pid JOIN conference AS t4 ON t3.cid = t4.cid WHERE t4.name = 'C' GROUP BY t1.name HAVING count(t3.pid) > 5`
D1	M	List the titles of publications published by author A.	`SELECT t3.title FROM author AS t1 JOIN writes AS t2 ON t1.aid = t2.aid JOIN publication AS t3 ON t2.pid = t3.pid WHERE t1.name = 'A'`
D2	M	List the names of organizations in continent C.	`SELECT name FROM organization WHERE continent = 'C'`
D3	H	List authors with more than 8 papers in conference C.	`SELECT t1.name FROM author AS t1 JOIN writes AS t2 ON t1.aid = t2.aid JOIN publication AS t3 ON t2.pid = t3.pid JOIN conference AS t4 ON t3.cid = t4.cid WHERE t4.name = 'C' GROUP BY t1.name HAVING COUNT(t3.pid) > 8`

Table 3.8: Tasks for the user study vs. PBE, with abbreviated foreign key names and literal values.

CHAPTER 4

Final Query Selection with Distinguishing Tuples

4.1 Introduction

Traditionally, querying databases has required knowledge of structured query languages such as SQL as well as an understanding of the database schema at hand. Users without knowledge of such structured query models can still specify a query by other means, such as natural language [5, 77], or query-by-example/query reverse engineering [49]. We collectively call these *oblique query specification (OQS)* systems, because they specify structured queries in only an oblique/indirect manner.

Users of OQS systems provide an incomplete and imprecise query specification. These systems must then translate this into a precise query matching the specification. Typically, OQS systems first formulate a set of precise *candidate queries (CQs)*, and then choose from among these alternatives. While many OQS systems can quickly narrow down to a small set of CQs, they often have to work hard to select the final target query from the set. Some systems may attempt to do so in an automated manner, using information such as the schema or logs, but eventually, OQS systems consult the user, whether proactively or as a last resort. Consider this *target query selection* example:

Example 4.1. *Sharon has been a car parts wholesaler in the USA for 15 years and has access*

to a relational database of part sales that a consulting firm created for her. After hearing recent news that tariffs would be enforced on goods flowing into and out of China, she wants to know which of her largest customers would be affected to inform them.

Unfortunately, she has little knowledge of SQL or of the database schema. As such, she uses a natural language interface (NLI) [5] on the database to issue the query: "What are the names and addresses of those in China who bought something worth more than $10,000 from us?"

Internally, the NLI tries its best to resolve the ambiguities in Sharon's query. In particular, "those in China" can refer to either customers or suppliers, and the amount "$10,000" can refer to various price fields. A few sample CQs are:

CQ1:
```
SELECT s.name, s.address
FROM supplier s
  JOIN partsupp ps ON ps.sid = s.sid
  JOIN part p ON p.pid = ps.pid
WHERE p.price > 10000
  AND s.address LIKE `%China%'
```

Meaning: Name and address of suppliers selling parts costing more than $10,000 with address containing substring 'China'.

CQ2:
```
SELECT c.name, c.address
FROM customer c
  JOIN nation n ON c.nid = n.nid
  JOIN order o ON o.oid = c.cid
WHERE o.price > 10000 AND n.nation = `China'
```

Meaning: Name and address of customers in China who made orders (i.e. collections of line items) of more than $10,000.

CQ3:
```
SELECT c.name, c.address
FROM customer c
```

```
  JOIN nation n ON c.nid = n.nid
  JOIN order o ON o.oid = c.cid
  JOIN lineitem li ON o.oid = li.oid
WHERE li.price > 10000 AND n.nation = `China'
```

Meaning: Name and address of customers in China who made an order with a line item costing more than $10,000.

A full list of the top 20 SQL CQs are directly displayed to Sharon, whose limited knowledge of SQL causes her to be overwhelmed by the options. She thus finds it difficult to select her target query from the list.

As demonstrated by this example, while users can issue query specifications on OQS systems as a "coarse-grained" filter to whittle down the universe of possible queries to a smaller set of CQs, there is still a need for a *"fine-grained" selection mechanism* for target query selection from this set.

Existing OQS systems sometimes provide such mechanisms, which are usually orthogonal to the CQ generation procedure. These include asking the user to manually examine the SQL syntax for each candidate query [10, 73], which is challenging for users unacquainted with SQL; examine query results when executed on synthetic data [44], which requires users to have schema knowledge; or put the burden on users to provide example output tuples [59] of their desired query.

Interaction Model We propose the *distinguishing tuple* interaction model to help users to select a target query from a set of CQs produced by OQS systems. The system suggests tuples from the result set of the CQs to the user and asks them whether their target query should contain it. The model aims to conserve user effort by *distinguishing multiple CQs at once* given user feedback on the suggested tuples.

The distinguishing tuple interaction model is *complementary* to most current models. Consider query-by-example/query reverse engineering [49] methods, which solicit exam-

	Column 1	Column 2	CQs
✗	Steeler Car Parts	555 China St, Pittsburgh, PA	1
✓	Beijing Auto Parts	Beijing, China	2, 3
	Great China Auto	Shanghai, China	2, 3
✗	Guangdong Auto	Guangzhou, China	2

Table 4.1: Example distinguishing tuple interaction.

ple tuples from the user: our interaction model leverages system-suggested tuples rather than user-suggested ones, and when both interaction models are used in tandem, users may opt at each iteration to either provide tuples or wait for the system to provide tuples.

For Example 4.1, the system would present the example tuples displayed in Table 4.1 and ask Sharon whether her desired query should produce each tuple. Sharon **rejects** (✗) the first tuple because the company is clearly not in China, eliminating CQ1. She also rejects the fourth tuple, knowing that Guangdong Auto only ever purchases small parts, and so eliminates CQ2. She **accepts** (✓) the second tuple, remembering that she sold an expensive part to Beijing Auto Parts earlier in the year. She **ignores** the third tuple because she can't precisely remember her interactions with that particular company. Sharon's feedback would then be evaluated by the system to eliminate all CQs except the target query CQ3.

The distinguishing tuple interaction model has several advantages over previous approaches. First, tuples are a ***common representation*** already used in various interaction models [13, 59] and ***requires no user expertise*** in SQL or the database schema. Second, a tuple can ***precisely distinguish two queries*** (so long as such a tuple exists) given a specific database instance. Finally, the interaction model ***reduces user effort*** by transferring the burden of suggesting examples in more traditional query-by-example or query reverse engineering approaches from the user to the system.

Of course, the effectiveness of the interaction model requires that the user knows both the structure (number and order of projected columns) of their desired output as

well as sufficient domain knowledge to provide feedback on output tuples. The former is reasonable to assume given that the user is the one who initiates the task on the OQS system. Our target user in this chapter is a domain expert, and the latter condition is trivially satisfied for such a user.

Technical Challenges We want to save user effort by arriving at their target query while displaying as few tuples as possible. This entails that we select the smallest set of tuples to whittle down the CQ set to the target query.

In addition, since the suggested tuples can only be retrieved by executing CQs on the database, this process may require the user to wait a long time for CQs to execute, depending on the size and schema of the database and the CQ workload. We aim to reduce the time to select a tuple by intelligently avoiding a full execution of all CQs.

In summary, our technical challenges are to: (1) *minimize the number of tuples needed to arrive at the target query*, and (2) *minimize the system time required to discover those tuples*.

Our Approach Minimizing the number of tuples presented to the user turns out to be NP-hard. Therefore, we devise a data structure, called *optimal split tree*, that can support good heuristics. The optimal split tree is a flowchart of potential tuples the system presents to the user depending on the user's feedback. We first present a *greedy algorithm* for constructing such a split tree. Then, we construct a novel data structure called the *Query Intersection Graph (QIG)* using information such as the *data types* and *intersecting values* of projected attributes in CQs. The QIG is used in *branch-and-bound* and *heuristic-based* variants of the algorithm, which provide runtime improvements.

Contributions We offer the following contributions:

- We introduce the *distinguishing tuple* interaction model to select a target query in a CQ set from OQS systems.

- We provide a *formal definition* of the MinDistTuples problem of minimizing the number of tuples in the distinguishing tuple model and a *proof of NP-hardness*.
- We develop *three different variants of a greedy algorithm* (GreedyAll, GreedyBB, and GreedyFirst) to solve the problem of minimizing user effort.
- We demonstrate through an *experimental evaluation* that our algorithms *reduce the number of tuples displayed to the user by up to 63% over state-of-the-art baseline approaches*.

In Section 4.2, we present an overview of the interaction model and formalize our problem. We introduce our solution strategy and algorithms in Section 4.3. In Section 4.4, we present our experimental evaluation. We describe related work in Section 4.5 and conclude in Section 4.6.

4.2 Overview

In this section, we provide an overview of the distinguishing tuple interaction model and a formal problem definition.

4.2.1 Interaction Model

Figure 4.1 displays an overview of the distinguishing tuple interaction model. The user begins by providing a "coarse-grained" specification of their target query to an OQS system. This initial specification can be made with any OQS system which will generate a finite CQ set, such as a natural language query, query-by-example, or query reverse engineering.

The system selects a tuple from the result set of the CQs, and presents it to the user. The user can either *accept*, *reject* or *ignore* the presented tuple. An accepted tuple is expected by the user in the output of their target query, while a rejected tuple is expected not

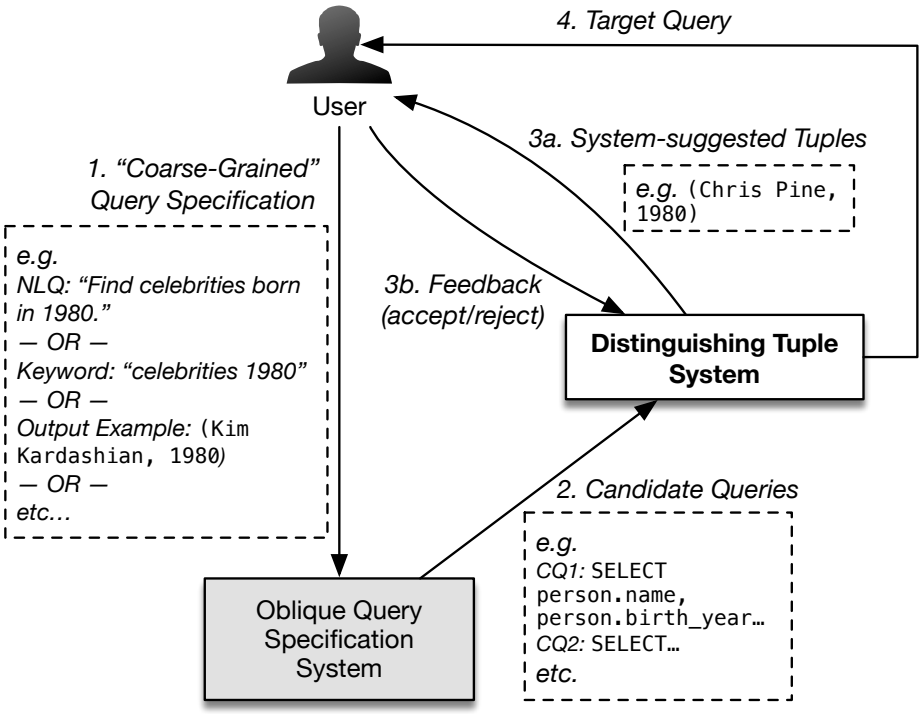


Figure 4.1: Overview of the interaction model.

to be in the output of the target query. If the user ignores a tuple, then an alternate tuple is displayed to the user. The system prunes the set of CQs according to the user's feedback, then again returns a tuple from the remaining CQs. This process iteratively continues until the system can arrive at a unique query satisfying all of the user's feedback[1].

4.2.2 Problem Definition

In this section, we introduce some necessary concepts, then formalize our problem definition. All concepts and definitions provided are in the context of a non-empty database instance D with a *fixed schema and fixed data contents*.

First, we define candidate queries:

Definition 4.1. *A **candidate query** (CQ) q is a relational query with:*

[1]If the user makes mistakes, there may be no final CQ or the final CQ may not match the user's intent. In such a case the user may either review their tuple feedback history to check for errors, or choose just to reissue their query on the OQS system.

- *a weight* $w(q) > 0$*;*
- *projected attributes* $\pi(q) = (\alpha_1, \ldots, \alpha_\omega)$*, where each attribute* $\alpha_i \in \pi(q)$ *has a type* $\tau(\alpha_i)$ *(e.g.* `str`*).*

The result set of tuples produced by a CQ q when executed on the fixed database D is denoted as $R(q)$.

The weight $w(q)$ of a CQ models the confidence that a certain CQ is the target query. Many OQS systems [8, 77] generate scores to rank CQs, and these scores may be used for $w(q)$. If OQS scores are unavailable, alternate sources of information such as a query log may also be used—e.g. to assign higher $w(q)$ for more frequently executed queries. In many cases, the $w(q)$ values may reflect probability values, but our definition does not require them to be so. If there are no helpful sources of weight information, then the system can assign an identical default weight to all CQs.

We denote a set of CQs by $\mathcal{Q} = \{q_1, \ldots, q_n\}$ and extend the notation of result sets and weights to CQ sets such that $R(\mathcal{Q})$ is defined as the union of all result sets of CQs in $\mathcal{Q}$ and $w(\mathcal{Q})$ is the sum of the weights of all the CQs. $\mathcal{Q}^t_\top$ is the subset of CQs in $\mathcal{Q}$ that produce the tuple t in their result sets, $\mathcal{Q}^t_\bot$ is the subset of CQs in $\mathcal{Q}$ that do not produce t in their result sets, and $\mathcal{Q}^t_\emptyset$ is the same as $\mathcal{Q}$ as $\emptyset$ constitutes a no-op. We also use $\mathcal{Q}^t$ as shorthand for $\mathcal{Q}^t_\top$.

A set of CQs is considered to be **equivalent** if all member queries produce the exact same result set with respect to the fixed database D. The order of projected columns in CQs also matters, i.e. if two CQs are identical but have the same projected columns in different order, we consider them distinct. The domain of all possible tuples is denoted $\mathbb{T}$.

Our goal is to *minimize the number of tuples presented to the user in the distinguishing tuple interaction model.* Given our setting where the target query is unknown a priori and can only be discovered by soliciting user feedback on tuples, we define a *distinguishing tuple set* as a set of tuples which uniquely identifies the target queries consistent with the user's feedback from a set of CQs:

Definition 4.2. *Given a CQ set $\mathcal{Q}$, a user function $\mathcal{U} : \mathbb{T} \longrightarrow \{\top, \bot, \emptyset\}$, and an equivalent set of target queries $\hat{\mathcal{Q}}$, a **distinguishing tuple set** for $\hat{\mathcal{Q}}$ on $\mathcal{Q}$ is a set of tuples $S = \{t_1, \ldots, t_m\}$ such that each tuple $t_i \in R(\mathcal{Q})$ and:*

$$\bigcap_{t_i \in S} \mathcal{Q}^{t_i}_{\mathcal{U}(t_i)} = \hat{\mathcal{Q}} \tag{4.1}$$

In Definition 4.2, we model the user as a function that takes a tuple as input and returns $\top$ (i.e. accept), $\bot$ (reject), or $\emptyset$ (ignore) as output. We use an equivalent set of target queries $\hat{\mathcal{Q}}$ instead of a single target query because the distinguishing tuple model is unable to distinguish two CQs that produce identical result sets, and in a fixed database setting we can consider such queries to be identical. Our model also requires that the result set of all queries in $\hat{\mathcal{Q}}$ are non-empty.

We now formalize our main problem:

Problem 4.1 (MinDistTuples)**.** *Given a set of CQs $\mathcal{Q}$ and an equivalent set of target queries $\hat{\mathcal{Q}} \subseteq \mathcal{Q}$ on a non-empty database instance D and a user function $\mathcal{U} : \mathbb{T} \longrightarrow \{\top, \bot, \emptyset\}$, find the smallest distinguishing tuple set S for $\hat{\mathcal{Q}}$ on $\mathcal{Q}$.*

Unfortunately, solving this problem, defined with respect to a *variable set of CQs* for a database with *fixed schema and contents*, is non-trivial; in fact:

Theorem 4.1. MinDistTuples *is NP-hard.*

Proof. Consider k-DistTuples, the decision problem variant of MinDistTuples. The problem is whether there exists a distinguishing tuple set S such that $|S| \leq k$.

First, we show k-DistTuples is in NP. If we have $\mathcal{Q}$ and a sequence of tuples S such that $|S| \leq k$, we run $\mathcal{U}(t_i)$ for each $t_i \in S$ and store the results. We then iterate through and execute each CQ $q_j \in \mathcal{Q}$, and add q_j to a set $\mathcal{Q}^*$ if q_j is consistent with $\mathcal{U}(t_i)$ for all $t_i \in S$. Specifically, a CQ q_j is consistent with $\mathcal{U}(t_i)$ if $q_j \in \mathcal{Q}^{t_i}_{\mathcal{U}(t_i)}$. If $\mathcal{Q}^*$ is comprised of equivalent CQs, S is a solution to the problem, and S is not a solution otherwise. This verification was performed in polynomial time, and therefore k-DistTuples is in NP.

Now, we demonstrate k-DistTuples is NP-hard by reducing the SetCover problem [37] to it in polynomial time. The SetCover problem is: given universe X and a family Y of subsets of X, a cover is a subfamily $C \subseteq Y$ of sets whose union is X. Is there a cover of size k or less?

Reduction: Let X, Y, C, k be an instance of SetCover. We create an instance of k-DistTuples as follows:

- Generate a CQ in $\mathcal{Q}$ for each element in $X = \{x_1, \ldots, x_n\}$, such that $q_i \in \mathcal{Q}$ "corresponds" to x_i. Add an additional CQ $\hat{q}$ to $\mathcal{Q}$, making $\mathcal{Q} = \{q_1, \ldots, q_n, \hat{q}\}$. The SQL for each query in $\mathcal{Q}$ is initially `SELECT c FROM t` where c is some column in table t in D.
- The equivalent set of target queries is: $\hat{\mathcal{Q}} = \{\hat{q}\}$.
- For each set $Y_j \in Y$ (where j is the index, starting at 1, of Y_j in Y), insert the data value j into column c in D. Then, for each $x_i \in Y_j$, edit the SQL of query q_i "corresponding" to x_i by appending a disjunctive predicate $c = j$ to the `WHERE` clause, i.e. `SELECT c FROM t WHERE ... OR c = j`. Finally, let $\hat{j} = |Y| + 1$ and insert data value $\hat{j}$ into column c in D, and append `OR c =` $\hat{j}$ to each query in $\mathcal{Q}$.
- Define $\mathcal{U}'$ such that for all $1 \leq j \leq |Y|$, $\mathcal{U}'((j)) = \bot$, where (j) is a tuple comprised of the single value j. Also, define $\mathcal{U}'((\hat{j})) = \top$.
- Create S by adding the tuple (j) to S for each $Y_j \in C$.

Forward direction: if C is in SetCover*, S is in k-*Dist-Tuples. According to our reduction, when all (j) tuples corresponding to each $Y_j \in C$ are passed into $\mathcal{U}'$, the result is $\bot$. None of these tuples belong to $R(\hat{q})$, and therefore all CQs will be eliminated when checking the Equation 4.1 condition except $\hat{q}$, and S is a solution to k-DistTuples.

*Reverse direction: if S is in k-*DistTuples*, then C is in* SetCover. Assume C is not in SetCover, i.e. there is a $x_i \in X$ not covered by C. In this case, when we check the

Equation 4.1 condition, more than 1 non-equivalent CQ remains: $\hat{q}$, and q_i. Therefore, S is not in k-DistTuples and by contraposition, the statement is true.

Since k-DistTuples is in NP and NP-hard, it is NP-complete. Therefore, its optimization variant, MinDistTuples, is NP-hard. □

4.2.3 Task Scope

While our general problem is not restricted to a specific query workload, we focus our optimization efforts on *conjunctive select-project-join queries without nesting or aggregation* due to their ubiquity. In addition, while one can conceive of an OQS system that generates a large number of CQs as output, most existing OQS systems such as natural language interfaces [5, 77] or sample-driven schema mapping systems [59] emit only a few final CQs, on the order of tens to at most a hundred CQs. As such, in this work we focus specifically on *assisting users in selecting target queries from CQ sets generated for human consumption* and leave the application of the distinguishing tuple model to larger-scale CQ sets and more complex query workloads for future work.

4.3 Algorithm

In this section, we introduce our overall solution strategies to tackle the NP-hard MinDistTuples problem of minimizing the number of tuples we present to the user.

4.3.1 Initial Approach

We first introduce a naïve approach, *TopWeight*. Given weight values for each CQ, *TopWeight* selects tuples from the highest-weighted CQ because a higher weight implies the CQ is more likely to be a target query, and thus the user is more likely to accept a tuple produced by that CQ. This can lead in turn to the elimination of many lower-weighted CQs.

At each iteration, *TopWeight* executes the highest-weighted CQ. It randomly selects a tuple from the results and runs a *verification query* on the tuple with each of the other CQs to check whether the tuple belongs to those CQs or not.

Definition 4.3. *A **verification query** $v(q,t)$ for a CQ q and a tuple $t = (t_1,\ldots,t_\omega)$ is the query q with a predicate $\alpha_i = t_i$ conjunctively added for each projected attribute $\alpha_i \in \pi(q)$.*

Example 4.2. *The verification query for the CQ* `SELECT a, b FROM table WHERE c = 42` *and the tuple* $(1,2)$ *would be* `SELECT a, b FROM table WHERE d = 42 AND a = 1 AND b = 2`.

The tuple is then presented to the user. If the tuple is accepted, then all CQs with an empty verification query result are removed from the CQ set. If it is rejected, then all CQs with a non-empty verification query result are removed. The process iterates until the target queries are found.

TopWeight, however, can perform poorly in the worst-case scenario. For example, consider a situation where the selected tuple from the top-weighted CQ is produced by all other CQs in the CQ set. In this case, user feedback on the tuple will not eliminate any CQs. Consequently, it is important to consider the expected number of CQs a tuple will eliminate before presenting it to the user.

4.3.2 Split Trees

We turn our attention to improving the *TopWeight* approach by developing a method to select tuples more intelligently.

First, we adopt the *split tree* [40] to represent the space of interactions in the distinguishing tuple interaction model. The split tree is a flowchart that models various possible interaction paths composed of system-suggested tuples and user feedback (i.e. accepting or rejecting the tuples; ignoring is omitted as it does not affect the CQ set). Formally:

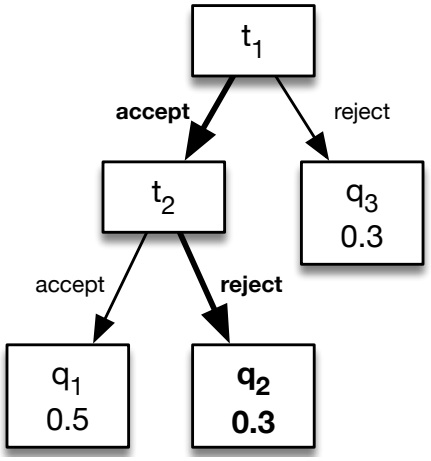


Figure 4.2: Example split tree. The bolded execution leads to q_2 as the target query.

Definition 4.4. *A **split tree** for CQ set $\mathcal{Q}$ is a rooted binary tree $\mathcal{T}$ in which each node v has a label $L(v)$ such that:*

- *Each set of equivalent CQs $\mathcal{Q}_i \subseteq \mathcal{Q}$ has exactly one corresponding leaf node $v_\ell \in \mathcal{T}$ labeled with $\mathcal{Q}_i$: $L(v_\ell) = \mathcal{Q}_i$ and each internal node $v_i \in \mathcal{T}$ is labeled with a tuple: $L(v_i) \in R(\mathcal{Q})$.*
- *Any CQ q in the left subtree of an internal node v_i produces the tuple $L(v_i)$ in its result set $R(q)$, while any CQ in the right subtree does not produce $L(v_i)$.*

As shown in Figure 4.2, a single instance of the distinguishing tuple interaction model can be mapped to a path from the root to the leaf node containing the target query. At each internal node, the left edge is taken if the user accepts the tuple or the right edge if the user rejects it. If we enumerated all root-to-leaf paths from all possible split trees, it would be equivalent to enumerating the entire search space of candidate distinguishing tuple sets for MinDistTuples.

4.3.2.1 Optimal Split Tree

One of the reasons why MinDistTuples is difficult is that the system has no way of knowing which CQs are target queries apart from a trial-and-error approach of feeding

tuples to the user. We attempt to tackle this challenge by minimizing the root-to-leaf path length for all CQs on a single split tree. The intuition is that the root-to-leaf path length represents the number of tuples displayed to the user to distinguish the CQ residing in that leaf node. Given that we do not know the target query a priori, the best we can do is to minimize this length for all CQs. Since the weights of CQs provide information on which CQs are most likely target queries, we include this and define the cost of a split tree as the *total weighted cost*:

$$c(\mathcal{T}) = \sum_{i=1}^{n} l_i w(q_i) \tag{4.2}$$

where l_i is the length of the path from the root to the leaf node labeled with q_i. While other cost functions such as the worst-case cost of any $q_i \in \mathcal{Q}$ are possible alternatives, we prefer the weighted cost because it takes into account any information provided by the user and/or OQS system to prioritize examining CQs with higher weights.

Using this cost metric, our strategy is to approximate MinDistTuples by discovering a single optimal split tree, consequently limiting the candidate distinguishing tuple sets to be explored to the root-to-leaf paths of this split tree:

Problem 4.2 (OptSplitTree)**.** *Given a set of CQs $\mathcal{Q}$, find the split tree $\mathcal{T}$ minimizing $c(\mathcal{T})$.*

While this problem is also demonstrated to be NP-hard [40], it allows us to move toward a feasible solution strategy.

4.3.3 Greedy Algorithm

The space of possible split trees that can be generated given a set of CQs is prohibitively large for most tasks, and so as a first step, we adopt the greedy approach described in [40] to approximate the optimal split tree.

Construction of the split tree happens recursively by selecting a tuple which creates the most balanced partition of the remaining CQs $\mathcal{Q}$. Formally, we find a tuple minimizing $|w(\mathcal{Q}^t) - w(\mathcal{Q} - \mathcal{Q}^t)|$. We add the tuple as a node, then split $\mathcal{Q}$ into subsets $\mathcal{Q}^t$ and $\mathcal{Q} - \mathcal{Q}^t$

on the left and right child node, respectively. The process is repeated on the subset CQs on each of the two child nodes until singleton sets result. We formalize the secondary problem of finding the next tuple for OptSplitTree according to this greedy strategy:

Problem 4.3 (OptTuple). *Given a set of CQs $\mathcal{Q}$, find:*

$$\arg\min_{t \in R(\mathcal{Q})} |w(\mathcal{Q}^t) - w(\mathcal{Q} - \mathcal{Q}^t)|$$

We can execute all CQs in $\mathcal{Q}$, then exhaustively scan all tuples in $R(\mathcal{Q})$ until we find one fulfilling OptTuple. We call this exhaustive approach GreedyAll, because it requires that the result set of all CQs be materialized before selecting even a single tuple. GreedyAll finds an exact, optimal solution to OptTuple, which is, accordingly, the next tuple to be selected for the greedy approach to solving OptSplitTree.

While GreedyAll is fitting for minimizing the number of tuples presented to the user, it requires an execution of all CQs which can potentially induce a long wait for the user, especially in the context of a large database or a large set of CQs. Consequently, we turn our attention to limiting the runtime of each iteration to reduce the user's wait time.

4.3.4 Partial Execution

One way to conserve time relative to the GreedyAll approach is by avoiding a full execution of all the CQs. Since our interaction model involves a human in the loop, we can accomplish this by only materializing the tuples in the split tree on paths corresponding to the user's feedback. In other words, instead of computing the full split tree, we leave some subtrees unrelated to the target queries unmaterialized, which in certain cases allows us to avoid executing CQs residing in those subtrees. For example, in the split tree in Figure 4.2, if q_1 is the target query, it is possible that we can present both t_1 and t_2 to the user by selecting them from the result set of q_1 without ever needing to execute q_2 or q_3.

To preserve the solution quality provided by the GreedyAll approach, we still want to find a tuple fulfilling OptTuple, yet without executing all CQs. The challenge is finding $\mathcal{Q}^t$, the subset of CQs in $\mathcal{Q}$ containing a tuple t, but we can only confidently do so after executing all CQs in $\mathcal{Q}$ that could possibly generate t. This entails that we know which pairs of CQs could possibly intersect—if there is a non-zero possibility of intersection between two CQs and we select a tuple t produced by one CQ, we must test whether t belongs to the result set of the other CQ to ensure correctness. The question is: *How can we discern whether the result sets of two CQs might intersect without fully executing them?*

4.3.4.1 Query Intersection Graph

We propose a data structure called the *Query Intersection Graph* (QIG) to model which CQs might have intersecting result sets, in which each node is a CQ and an edge exists between two nodes if there is *any possibility that the two CQs' result sets intersect.*

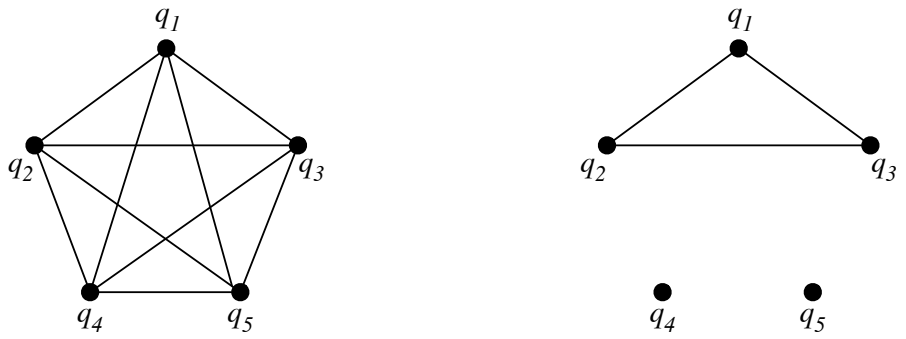
Definition 4.5. *The **query intersection graph (QIG)** for a set of CQs $\mathcal{Q}$ and information sources $\mathcal{I}$ is a graph $\mathcal{G} = (V, E)$ such that:*

- *Each query $q \in \mathcal{Q}$ has a corresponding node $v \in V$.*
- *An edge $e \in E$ exists between two nodes if their corresponding CQs have any possibility of intersection given information sources $\mathcal{I}$.*

Example 4.3. *Consider that an OQS system produces 5 CQs on a movie database, and q_1 is the target query:*

q_1: `SELECT p.name, m.title FROM person p, cast c, movie m WHERE m.mid = c.mid AND c.pid = p.pid`

q_2: `SELECT p.name, m.genre FROM person p, cast c, movie m WHERE m.mid = c.mid AND c.pid = p.pid`



(a) Initial state.

(b) With additional info.

Figure 4.3: Example query intersection graphs (QIG).

q_3: `SELECT p.name, m.country FROM person p, directed d, movie m WHERE m.mid = d.mid AND d.pid = p.pid`

q_4: `SELECT p.name, m.year FROM person p, cast c, movie m WHERE m.mid = c.mid AND c.pid = p.pid`

q_5: `SELECT p.name, m.budget FROM person p, cast c, movie m WHERE m.mid = c.mid AND c.pid = p.pid`

Without executing any of the CQs and with no external information, the initial QIG for Example 4.3 is a fully connected graph as in Figure 4.3a. In this state, the QIG conveys that any tuple produced in one of the CQs could potentially be produced by any of the other 4 CQs. In this situation, we would be required to execute all 5 CQs before being able to discern $\mathcal{Q}^t$ for any tuple, and consequently, we would be unable to confidently select any tuple satisfying OptTuple.

Now let's say that without executing the CQs, we gain some information (we elaborate more on specific information sources in Section 4.3.4.3) that (q_1, q_2), (q_1, q_3), (q_1, q_4), and (q_2, q_3) are the only pairs of CQs whose result sets could possibly intersect, generating the QIG in Figure 4.3b.

We now develop some intuition for how we might consume such a QIG to avoid executing all the CQs. In particular, we consider the *maximal cliques* of a QIG. In our example, if we execute the CQs in the maximal clique $\{q_1, q_2, q_3\}$ and find a tuple produced by *all* of the 3 CQs, *we do not need to execute any further CQs* to see if the tuple belongs to their result sets because the QIG tells us that at least one of the CQs is disjoint from each of q_4 and q_5. We posit, therefore, that executing in batches of maximal cliques seems a possible way to limit the number of CQs executed while guaranteeing that we can find the value of $\mathcal{Q}^t$ for tuples produced by all CQs in the maximal CQ. Consequently, we formally state:

Theorem 4.2. *A tuple t belonging to the result set of all CQs in a maximal clique $\mathcal{M}$ of the QIG is guaranteed not to belong to the result set of any CQ outside $\mathcal{M}$.*

Proof. Theorem 4.2 follows directly from the definition of a maximal clique, since there is no CQ outside a maximal clique which intersects with all the CQs in the maximal clique. Consequently, there can be no tuple that belongs to the result sets of all the CQs in the maximal clique but also belongs to the result set of a CQ outside the clique. □

If our goal is to minimize the number of CQs executed, one might ask why we can't just execute a batch of fewer queries than a full maximal clique. For our example, we can consider the non-maximal clique $\{q_1, q_2\}$, which is a subset of the maximal clique $\{q_1, q_2, q_3\}$. The problem is that if we execute this non-maximal clique, we find that even if we find a tuple which belongs to both q_1 and q_2, the QIG tells us we must still examine q_3 to see if the tuple is produced by q_3. The same goes for any tuple which belongs to only one of q_1 or q_2. This is captured in the following:

Theorem 4.3. *A tuple t belonging to the result set of all CQs in a clique $\mathcal{C}$ of the QIG can only occur in a CQ $q \notin \mathcal{C}$ if $C \cup \{q\}$ comprises a clique in the QIG.*

Proof. Assume there exists a tuple t belonging to all the result sets of CQs in a clique $\mathcal{C}$ and also to the result set of $q^* \notin \mathcal{C}$. Also assume that $C^* = C \cup \{q^*\}$ does not comprise a clique in the QIG. The fact that C^* does not comprise a clique in the QIG means that there

exists at least one $q_i \in C$ such that q_i has no edge with q^* in the QIG, which by definition means that $R(q_i) \cap R(q^*) = \emptyset$, which contradicts our first assumption of the existence of t. □

Theorem 4.3 states that the only CQs we need to execute in addition to the CQs in a clique to find $\mathcal{Q}^t$ are CQs which form a larger clique when added to the original clique. In other words, given a tuple t belonging to all CQs in a clique $\mathcal{C}$, we can find all the CQs t belongs to in $\mathcal{Q}$ by simply checking any CQs which are part of any maximal cliques subsuming $\mathcal{C}$.

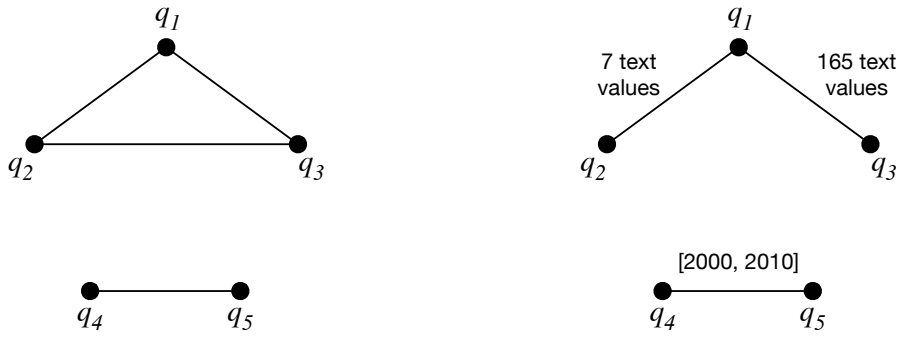
4.3.4.2 Position-wise QIGs

When constructing the QIG, we first consider each projected attribute position independently. This enables us to save effort by batch-processing CQs which have the same projected column at a given position. In Example 4.3, position 1 is comprised of the single attribute {`person.name`}, and position 2 contains attributes {`movie.title`, `movie.genre`, `movie.country`, `movie.year`, `movie.budget`}. Given this, we create *position-wise QIGs* which only consider the attributes at a specific position. Like full QIGs, given no information, a position-wise QIG is fully connected as in Figure 4.3a.

Definition 4.6. *A **position-wise QIG** for a set of CQs $\mathcal{Q}$, information sources $\mathcal{I}$, and projected attribute position $k \in \mathbb{N}$ is a graph $\mathcal{G}_k = (V_k, E_k)$ such that:*

- *Each query $q \in \mathcal{Q}$ has a corresponding node $v \in V_k$.*
- *An edge $e \in E_k$ exists between two nodes if their corresponding CQs have any possibility of intersection at projected attribute position k given $\mathcal{I}$.*

Position-wise QIGs can be merged into a full QIG by examining each pair of CQs in each position-wise QIG and adding an edge to the full QIG only if all of the position-wise QIGs have an edge between that pair of CQs. The following theorem formalizes this relationship:



(a) After data types

(b) After intersecting values

Figure 4.4: Position 2 QIG for Example 4.3.

Theorem 4.4. *If at least one position-wise QIG has no edge between two CQs, then the full QIG has no edge between the two CQs.*

Proof. Assume there is a position-wise QIG at projected attribute position i for which two CQs q_1 and q_2 have no shared edge. This means that no tuple in $R(q_1)$ produces the same value as a tuple in $R(q_2)$ at position i, and consequently it follows that $R(q_1) \cap R(q_2) = \emptyset$ and there is no edge between q_1 and q_2 in the full QIG. □

Our formulation of the construction of QIGs is a *subtractive* rather than an additive process, where we begin with a complete graph and remove edges based on information sources rather than taking a set of nodes and adding edges to it. As an alternative, we could construct the complement graph of a QIG in an additive process, where an edge represents that two CQs are disjoint, and consider the independent sets in the resulting complement graph instead of cliques.

4.3.4.3 Information Sources

The QIG has an edge between two CQs if there is any possibility of them producing the same tuple given our knowledge about the CQs. Consequently, we can eliminate edges if information is provided guaranteeing that the CQs are disjoint. We propose two specific

information sources that can be used to construct the QIG.

Data Types One information source to consider is the *data types* at each position. For position 2 in Example 4.3, `movie.year` and `movie.budget` are numeric attributes, while the other three are text attributes. For the position-wise QIG for position 2, we can eliminate edges between the numeric and text attributes because they will never produce the same value at that position in a tuple[2], resulting in the position-wise QIG in Figure 4.4a. We can then merge the position-wise QIGs for position 1 (a complete graph because all CQs share the same projected attribute) and position 2 into a full QIG following Theorem 4.4. For our example, the full QIG given data type information will be isomorphic to Figure 4.4a.

Full QIGs generated using only data type information are guaranteed to be composed of strongly connected components (one for each distinct list of projected data types), and the problem of finding all maximal cliques is trivially reduced to finding each of the strongly connected components and can be done in $O(|\mathcal{Q}|)$ time.

Intersecting Values The other information source we consider is the *intersecting values* of the attributes at the position. We introduce a data structure called the *Attribute Intersection Graph* (AIG) to store this information.

Definition 4.7. *The **attribute intersection graph (AIG)** for database D is a graph $\mathcal{A} = (V_{\mathcal{A}}, E_{\mathcal{A}})$ such that:*

- *Each attribute α in D has a corresponding node $v \in V_{\mathcal{A}}$.*
- *An edge $e \in E_{\mathcal{A}}$ exists between two nodes if their corresponding attributes have any intersecting values. Each edge also has metadata $m(e)$ storing the intersecting values of the attributes, as a closed interval range $m(e) = [a, b]$ for numeric attributes and a set of values $m(e) = \{c_1, \ldots, c_m\}$ for text attributes.*

[2]We assume that the database is strongly typed, where a numeric value in a text attribute is distinct from the same value in a numeric attribute (e.g. "4" ≠ 4).

We construct the AIG in an offline process by computing the intersecting values of each pair of attributes and storing the resulting graph to disk.

For Example 4.3, let's imagine we consult the AIG to find that for the text attributes at position 2, `movie.title` intersects with both `movie.genre` and `movie.country`, but `genre` is disjoint from `country`. For the numeric attributes, `movie.year` has values in [1953, 2010] and `movie.budget` has values in [2000, 52,000,000]. These attributes intersect in the range [2000, 2010]. The resulting position-wise QIG for position 2 is shown in Figure 4.4b.

Again, we follow Theorem 4.4 to merge each of the position-wise QIGs to construct a full QIG. In our running example, the full QIG will be isomorphic to the position 2 QIG because the position 1 QIG is fully connected and adds no additional information. The resulting maximal cliques in the full QIG are $\{q_1, q_2\}$, $\{q_1, q_3\}$, and $\{q_4, q_5\}$.

The offline AIG approach is only compatible with CQs that project the raw, untransformed values of the database instance (e.g. select-project-join queries). We leave the adaptation of these techniques to more complex queries such as aggregate or nested queries for future work.

4.3.4.4 Branch and Bound Algorithm

We present a branch and bound algorithm, GreedyBB, which aims to find a tuple satisfying OptTuple while executing as few CQs as possible. GreedyBB uses the QIG and Theorems 4.2 and 4.3 to *execute batches of CQs one maximal clique at a time*.

Branch and bound is a technique commonly used for NP-hard problems, where the space of candidate solutions to an optimization problem is constructed as a rooted tree using two operations: *branch*, which recursively splits the search space into smaller spaces, and *bound*, which returns the lower bound of any candidate solution and its descendants. Then, a top-down recursive search is performed which prunes any branches whose lower bound is higher than an already-explored candidate solution.

Algorithm 8 GreedyBB

function GreedyBB($\mathcal{Q}$,G)
  Init $\mathcal{P}$ as priority queue
  $\mathcal{C} \leftarrow$ *FindMaxCliques*$(\mathcal{Q}, G)$
  for $\mathcal{C}_i \in \mathcal{C}$ **do**
    Add $(bound(\mathcal{C}_i), \mathcal{C}_i, \mathcal{C}_i)$ to $\mathcal{P}$
  $\hat{T} \leftarrow \emptyset$
  $\hat{v} \leftarrow \infty$
  while $\mathcal{P} \neq \emptyset$ **do**
    Pop next $(\mathcal{B}, \mathcal{S}, \mathcal{X})$ from $\mathcal{P}$
    if $\mathcal{B} \geq \hat{v}$ **then continue**
    ExecuteBatch$(\mathcal{X})$
    $T \leftarrow \{t : t \in R(\mathcal{X}) \wedge \mathcal{Q}^t = \mathcal{S}\}$
    if $T \neq \emptyset$ **then**
      $\hat{T} \leftarrow T$
      $\hat{v} \leftarrow |w(\mathcal{S}) - w(\mathcal{Q} - \mathcal{S})|$
    $U \leftarrow \{t : t \in R(\mathcal{X}) \wedge \mathcal{Q}^t \subset \mathcal{S}\}$
    if $U \neq \emptyset$ **then**
      Add $branch(\mathcal{Q}, \mathcal{C}, U)$ to $\mathcal{P}$
  return $\hat{T}$

Algorithm 8 shows the GreedyBB approach. It takes a CQ set $\mathcal{Q}$ and the QIG G of the set, and returns a set of tuples fulfilling OptTuple. A priority queue $\mathcal{P}$ stores the search space. We assume that $\mathcal{P}$ does not allow duplicate items. Each item in $\mathcal{P}$ is a triple $(\mathcal{B}, \mathcal{S}, \mathcal{X})$, where $\mathcal{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_k\}$ is a set of CQs forming a clique in the QIG and $\mathcal{X}$ is the set of CQs that need to be executed before we are able to find $\mathcal{Q}^t$ for any tuple in $\mathcal{S}$. $\mathcal{P}$ is sorted in ascending $\mathcal{B} = bound(\mathcal{Q}, \mathcal{S})$ order, where *bound* is a function defining the lower bound for the OptTuple objective:

$$bound(\mathcal{Q}, \mathcal{S}) = \begin{cases} w(\mathcal{Q} - \mathcal{S}) - w(\mathcal{S}) & \text{if } w(\mathcal{Q} - \mathcal{S}) \geq w(\mathcal{S}) \\ min\{\, w(\mathcal{S}) - w(\mathcal{Q} - \mathcal{S}), & \text{otherwise} \\ \quad bound(\mathcal{Q}, \mathcal{S} - \{\mathcal{S}_1\}), \ldots, \\ \quad bound(\mathcal{Q}, \mathcal{S} - \{\mathcal{S}_k\})\} \end{cases}$$

The first case in *bound* considers when the weight for $(\mathcal{Q} - \mathcal{S})$, the CQs excluded from $\mathcal{S}$, exceeds that of $\mathcal{S}$. In this case, the best (i.e. smallest) possible value we can achieve for

our OptTuple objective is in the case when a tuple belongs to all CQs in $\mathcal{S}$, as this will maximize the $w(\mathcal{Q}^t)$ term in the OptTuple objective while shrinking the $w(\mathcal{Q} - \mathcal{Q}^t)$ term. Consequently, we return this minimal value for the first case.

In the second case, there is a higher weight for the $\mathcal{S}$ set than the $(\mathcal{Q} - \mathcal{S})$ set, meaning that there may be a tuple which belongs to a *subset* of $\mathcal{S}$ which minimizes the OptTuple objective. This is because a tuple belonging to a subset of $\mathcal{S}$ could potentially reduce the $w(\mathcal{Q}^t)$ term compared to a tuple that belongs to all CQs in $\mathcal{S}$, leading to a smaller objective value. However, doing this could potentially also cause $w(\mathcal{Q}^t)$ to be larger than $w(\mathcal{Q} - \mathcal{Q}^t)$, which is why a recursive call is required to evaluate both the first and second case in the bound function for any subset of $\mathcal{S}$.

All maximal cliques on the QIG are computed using an optimized version of the Bron-Kerbosch algorithm [16,68] and added to $\mathcal{P}$ (Line 3). In every iteration, the highest priority item is popped from $\mathcal{P}$, then its $\mathcal{B}$ value is checked against the current upper bound $\hat{v}$ (Line 10), and the current branch is pruned if it does not pass. If it passes, we execute the batch of CQs $\mathcal{X}$ (Line 11). *ExecuteBatch* executes all unexecuted CQs in the batch on the database and retrieves cached result sets for already-executed CQs. Then, we find T, which is the set of all tuples which belong to exactly the CQs in $\mathcal{S}$ (Line 12). Because of Theorem 4.3, we can find the exact value of $\mathcal{Q}^t$ only by checking queries in $\mathcal{X}$. If T is non-empty, we can update the current best solution $\hat{T}$ and upper bound $\hat{v}$. We then find U, the set of all tuples which belong to a proper subset of the CQs in $\mathcal{S}$ (Line 16), and we *branch* (Line 18), which produces all smaller cliques $\{(\mathcal{B}_1, \mathcal{Q}^{t_1}, \mathcal{X}_1), \ldots, (\mathcal{B}_k, \mathcal{Q}^{t_k}, \mathcal{X}_k)\}$ for each $t_i \in U$, where each $\mathcal{B}_i = \mathit{bound}(\mathcal{Q}, \mathcal{Q}^{t_k})$, and, due to Theorem 4.3, each $\mathcal{X}_i$ is formed of the union of all maximal cliques $\mathcal{C}_j \in \mathcal{C}$ such that $\mathcal{Q}^{t_k} \subset \mathcal{C}_j$. We continue the loop until $\mathcal{P}$ is empty. The returned $\hat{T}$ is the solution to OptTuple.

Algorithm 9 GreedyFirst

```
1: function GreedyFirst(Q,G)
2:     C ← FindMaxCliques(Q, G)
3:     Sort C by bound ascending
4:     for i = 1,..., m do
5:         ExecuteBatch(C_i)
6:         S ← C_1 ∪ ... ∪ C_i
7:         T ← {t : t ∈ R(S) ∧ (∀j > i, S^t ⊄ C_j)}
8:         if T ≠ ∅ then
9:             return arg min_{t∈T} |w(Q^t) − w(Q − Q^t)|
```

4.3.4.5 Heuristic Approach

While GreedyBB is expected to make a runtime improvement over the GreedyAll approach, it still adheres to producing an exact solution to OptTuple. In our interaction model, it may be advantageous to produce a tuple as fast as possible to the user by sacrificing exactness and offering an approximate solution to OptTuple.

We propose a heuristic to return a reasonable approximation to OptTuple in minimal time. This approach is called GreedyFirst (Algorithm 9). In this algorithm, we still calculate the maximal cliques of the QIG as in GreedyBB, but we execute the maximal clique with lowest bound first and use Theorem 4.3 to find the set of tuples T for which we can compute Q^t without checking any CQs outside already-executed cliques (Line 7), and then return the tuple within T which minimizes the OptTuple objective (Line 9).

Depending on the characteristics of the CQ set, this heuristic approach could save execution time for earlier iterations of OptSplitTree when most CQs have yet to be executed.

4.4 Evaluation

We investigate the following research questions:

- **RQ1:** Do our algorithms minimize the number of tuples presented to the user?

- **RQ2:** How are our algorithms affected by the reliability of provided CQ weights?
- **RQ3:** What are the runtimes of our algorithms?

4.4.1 Experimental Setup

We consider a setting in which a domain expert attempts to select a target query from a set of candidate queries generated from an OQS system. We assume that the user knows the format of output desired from the target query and is also able to correctly accept or reject any tuples presented by the system. If these assumptions are met, the user is just an "automaton" and does not need to exercise any judgment. Therefore, we lose nothing by using a simulated user in our studies, which we do. Whether these assumptions are satisfied by real users is primarily determined by their domain knowledge. But that is a factor completely controlled by study design in a lab user study. If we want to understand user domain knowledge, we would have to do studies in the field. Since we did not have the resources for a field study, we chose to perform a simulated user study, which would be just as informative as a user study in the lab or on Amazon Turk.

4.4.1.1 Procedure

The input for each task was a set of CQs with a single target query. For each iteration of the task, the system selected a tuple from the result sets of the CQs and presented it to the simulated user, which accepted or rejected the tuple. The system eliminated CQs given the user's feedback, then continued another iteration. The task terminated when the system narrowed down the CQ set to a single CQ, which was returned as the target query.

We compared our algorithms, GreedyAll (All for short), GreedyBB (BB), and GreedyFirst (First) to *TopWeight* (*TopW*) and the L^1S algorithm [12, 13], which first materializes all candidate tuples, then selects tuples which eliminate the greatest number of candidate tuples. This differs from our algorithms, which select tuples that eliminate the greatest

Dataset	Database		Tasks		CQs / Task	
	Engine	Size	Easy	Hard	Mean	Max
Mondial	MyISAM	1.8 MB	45	23	92.74	1711
IMDB	MyISAM	2.4 GB	57	7	30.69	486
Yelp	InnoDB	2.7 GB	36	0	6.92	33

Table 4.2: Datasets used in our evaluation.

number of *candidate queries*. For each task, we averaged the results of 5 trials with each algorithm.

We did not compare against the bottom-up and top-down algorithms from [12, 13] because they were designed only for join predicate workloads. We also did not evaluate against L^2S as it leveraged a similar approach to L^1S yet was demonstrated to be an order of magnitude slower than L^1S. In addition, while a query-by-example and query reverse engineering systems [49] enable *users* to provide examples, we do not compare against them as their contributions are *complementary* to our approach. A user may leverage such systems by providing any examples they can think of off the top of their head, and any candidate queries produced can then be passed into our approach to select the final target query.

All evaluations were performed on a machine with a 2.8 GHz AMD Opteron 6320 processor, 503 GB RAM, and a 27.3 TB solid-state drive, running Ubuntu 16.04 and MySQL 5.7.23 with a disabled query cache, a 16 MB key cache for MyISAM databases, and a 1 GB InnoDB buffer pool which was reset before running each algorithm on each dataset. We set a timeout of 20 seconds on every query issued to MySQL.

4.4.1.2 Datasets

We used a set of benchmarks reflecting the scenario where an OQS system had already been engaged to produce a set of CQs. Table 4.2 summarizes some statistics for each of these datasets.

The Mondial dataset [35] is comprised of target queries which were randomly generated from the schema of the Mondial database [50]. The output of the target queries was then examined and reverse-engineered to generate constraints which were executed on the Beaver [35] system to generate a set of CQs for each target query. Every CQ in Mondial is a project-join query with a single join of two relations.

IMDB and Yelp are both introduced by [77]. Each dataset contains a SQL database and a corresponding set of natural language query tasks. We executed the natural language queries for each task using a natural language interface from [5] to produce a set of candidate queries which vary in terms of selected projections, predicates, and join paths. We only retained tasks with conjunctive select-project-join queries, removed duplicate tasks, and modified tasks for which the target query produced an empty set. For each task, we manually annotated the correct target query.

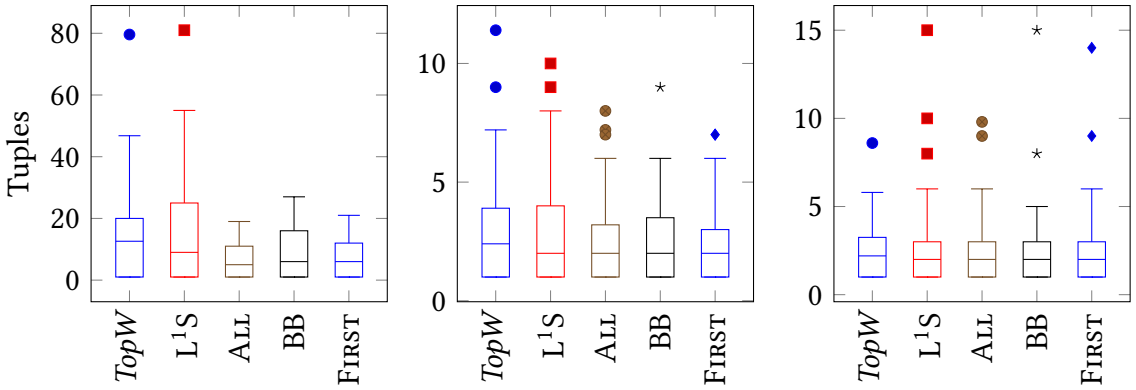
We limited each CQ set to have a maximum total query execution time of 15 minutes, where timed out CQs were assigned the timeout limit as their execution time. We eliminated non-target query CQs from each task until the total query execution time was below 15 minutes.

4.4.1.3 Task Difficulty

As a rough measure of the difficulty of a task, we introduce the *target query confusion (TQC)* metric given a CQ set Q and a target query $\hat{q} \in Q$:

$$TQC(Q, \hat{q}) = 1 - \frac{1}{\sum_{q \in Q} \frac{|R(q) \cap R(\hat{q})|}{|R(\hat{q})|}} \tag{4.3}$$

The $\frac{|R(q) \cap R(\hat{q})|}{|R(\hat{q})|}$ value in the denominator measures how many of $\hat{q}$'s output tuples are included in a CQ q's result set. If q produces all the tuples in $\hat{q}$, this value will be 1, and if q produces none of them, the value will be 0. This value is summed over all CQs to produce a rough measure of how many CQs might be confused with the target query. The



(a) Mondial (b) IMDB (c) Yelp

Figure 4.5: # tuples presented for easy tasks (TQC ≤ 0.75).

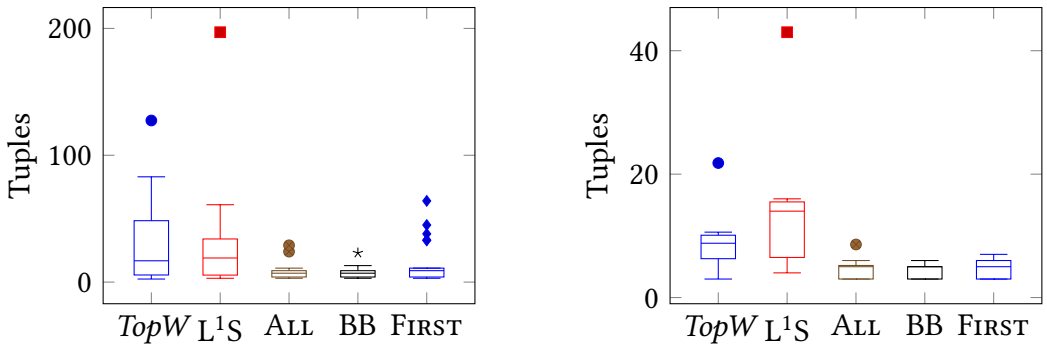
Dataset	*TopW*	L^{1}S	All	BB	First
Mondial	0.35	0.36	**0.24**	0.25	0.26
IMDB	0.26	0.28	0.25	0.25	**0.24**
Yelp	0.43	0.45	0.43	**0.40**	0.44

Table 4.3: Mean ratio of tuples to CQ count.

reciprocal of this number of CQs is the probability of selecting the target query correctly from the set of confusing CQs. This probability is subtracted from 1 to reflect the chances of selecting the wrong query as the target query.

When we calculated the TQC values for tasks in our dataset, we enforced a 100 second timeout when executing the query to calculate the fraction in the denominator for each CQ. If this query timed out, we sampled 1000 tuples from the target query and ran a verification query for each of these tuples on the CQ to calculate the proportion of the sampled tuples that would be produced by the CQ. This proportion was then used as an estimate for the fraction.

We categorized tasks as easy if they had a TQC value ≤ 0.75, and hard otherwise. We display the number of easy and hard tasks for each dataset in Table 4.2.



(a) Mondial (b) IMDB

Figure 4.6: # tuples presented for hard tasks (TQC > 0.75).

4.4.2 Benefit of Distinguishing Tuple Model

To evaluate the benefit of the distinguishing tuple interaction model, we compared it against a typical classic interaction model. The classic model for many OQS systems [10, 73] presents a list of the SQL for each of the CQs to the user and asks them to examine it. When the CQs have equal weights, *the average case is for the user to examine half the CQs.*

While it is difficult to make a direct quantitative comparison between our interaction model and the classic model, we measured the ratio of tuples presented to the user with the number of CQs in each task. While in reality we believe that examining a tuple requires less effort and expertise than examining SQL syntax, this ratio metric treats examining tuples and SQL as requiring equal effort.

In Table 4.3, we present the ratio measured on our tasks in the equal weight setting. The ratio never exceeded 0.5 for any algorithms, meaning that our interaction model only needed to display less than half as many tuples as the number of CQs in the task on average. Our algorithms performed well on Mondial in particular, driving down the value to as little as 0.24. Given that a user would be expected to examine half (i.e. 0.5) of the CQs in the classic model and assuming that a tuple is easier to examine than the full SQL of a CQ, these results indicate that **the distinguishing tuple model requires significantly less user effort than the classic model**.

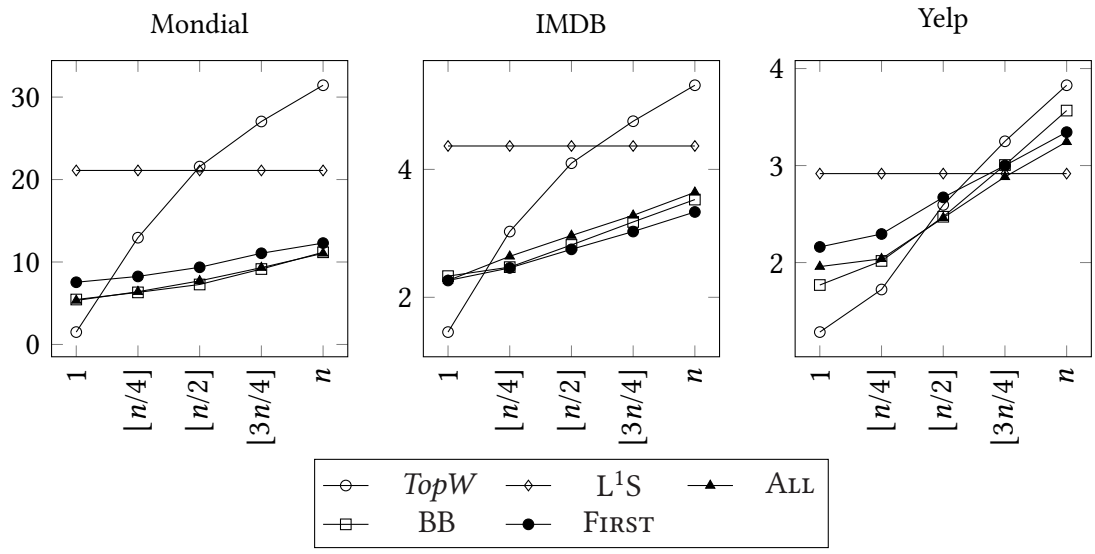


Figure 4.7: Mean tuples displayed per task depending on the enforced target query ranking.

4.4.3 User Effort

To answer **RQ1**, we measured the number of tuples that needed to be displayed to the user to find the target query.

4.4.3.1 Equal CQ Weights

We first considered a scenario where all CQs have an equal weight $w(q) = 1$. This reflects a scenario where there is no reason for one CQ to be preferred over another. Figures 4.5 and 4.6 respectively display the number of tuples taken on easy and hard tasks for each dataset. The box-and-whisker plots display the minimum, first quartile, median, third quartile, and maximum values, along with any outliers (values greater than the upper quartile by at least 1.5 times the interquartile range or lesser than the lower quartile by at least that amount) as individual points.

On Mondial, our algorithms demonstrated a significant improvement over *TopW* and L^1S. The mean number of tuples required for *TopW* and L^1S on the entire Mondial dataset were 19.57 and 21.10 tuples respectively, compared to 7.24 for All, 8.22 for BB, and 9.71

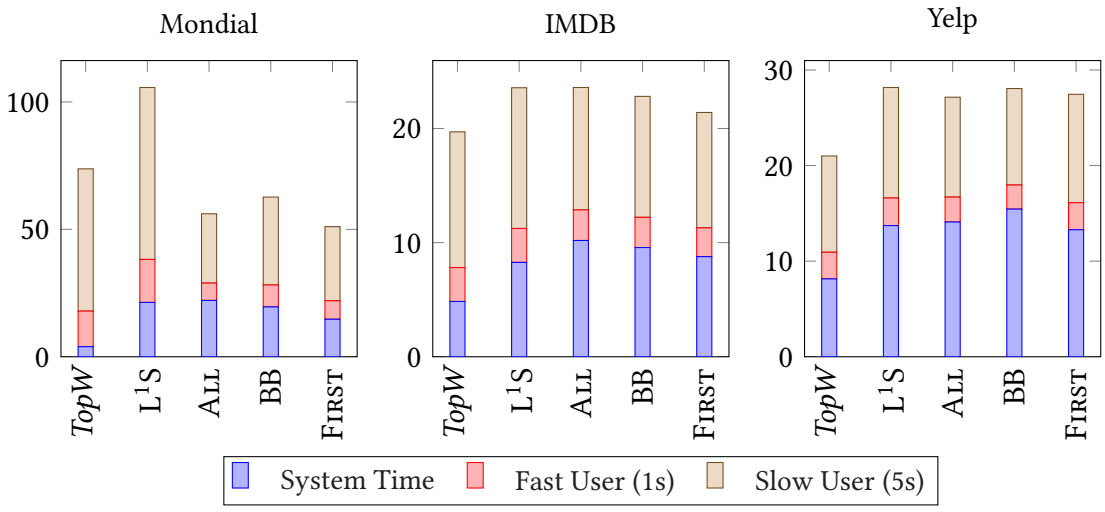


Figure 4.8: Mean total runtime (s) on easy tasks (TQC ≤ 0.75).

for First, demonstrating **a minimum of > 50% reduction in user effort** from using our algorithms over *TopW*, with **up to a 63% reduction** for All in particular.

For hard tasks in IMDB, we observed a similar improvement, where the median number of tuples was 8.8 for *TopW* and 14.0 for L^1S while the maximum (excluding outliers) for all our algorithms peaked at 7.0. The mean tuples were 9.49 for *TopW* and 15.0 for L^1S, while our algorithms produced 5.11 for All, 4.71 for BB, and 5.0 for First, again **demonstrating at least a 46% reduction in effort using our algorithms**.

For the easy tasks in IMDB and Yelp, all algorithms performed comparably. This follows from the fact that the mean TQC value for easy tasks in Mondial was 0.33, while the mean TQC for easy tasks in the other datasets were 0.16 for IMDB and 0.18 for Yelp. Though the TQC metric is a rough metric for task difficulty, these numbers indicate that the easy tasks in IMDB and Yelp were easy enough that there was not much room for improvement.

In summary, **our algorithms performed similarly to *TopW* and L^1S for easy tasks and performed significantly better for harder tasks**. This is because our algorithms are optimized to tuples which eliminate the most CQs, while L^1S is optimized to return tuples which eliminate the greatest number of candidate tuples after materializing

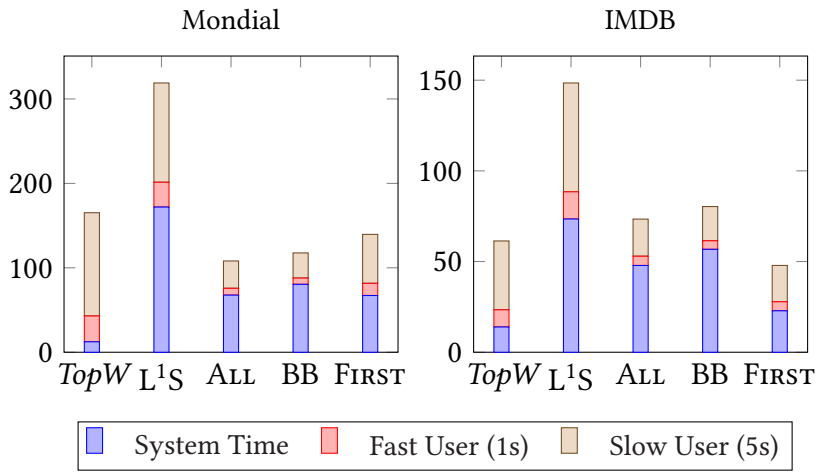


Figure 4.9: Mean total runtime (s) on hard tasks.

the result sets of all CQs—a fine strategy for join query workloads, but a suboptimal one for general CQ workloads. *TopW* is ineffective as it constitutes a random strategy when CQs have equal weights.

4.4.3.2 Unequal CQ Weights

To answer **RQ2**, we considered the effect of an OQS system or user assigning unequal weights to CQs. We implemented a weighting scheme where the k-ranked CQ was assigned $w(q) = n - k + 1$, making the top-ranked CQ $w(q) = n$ and the last-ranked CQ $w(q) = 1$. Then we tested scenarios where we assigned the target query rank 1, $\lfloor n/4 \rfloor$, $\lfloor n/2 \rfloor$, $\lfloor 3n/4 \rfloor$ and n respectively for each task, while all other CQs were randomly ranked. The scenario where the target query was ranked 1 reflected the best case where the target query was correctly assigned the highest weight, while a ranking of n reflected the worst case where the target query was assigned the lowest weight.

Figure 4.7 displays the effect of changing the target query ranking on the number of tuples. L^{1}S is a horizontal line because the algorithm does not take weights into consideration. For all 3 datasets, the trend for *TopW* had a steeper slope than all of the other algorithms. The difference was most evident in Mondial and IMDB, which had a high

proportion of hard tasks. For Mondial, regardless of the weights, our 3 algorithms required less than 13 tuples per task. On the other hand, in the worst-case scenario, the naïve approach *TopW* required around 3 times the tuples of our algorithms, asking the user to view upwards of 30 tuples on average.

These results indicate that when compared to the baseline approach, **our algorithms are resilient even when the assigned weights for CQs are unreliable**.

4.4.4 Runtime

To answer **RQ3**, we measured the mean total runtime of each algorithm over all tasks in the equal weight setting. Total runtime is comprised of two components: *system time*, which includes database query time, algorithm time and other overhead; and *user time* per iteration. We consider two user scenarios: a fast user with a 1 second response time per iteration, and a slow user with a 5 second response time.

Figure 4.8 displays the results for easy tasks. *TopW* has the lowest system time because it executes only a single CQ per iteration. For a fast user, *TopW* exhibits the lowest runtime for all three datasets, though often requiring that users provide feedback on more tuples. For a slow user, however, results are mixed, with our three algorithms performing better on Mondial and slightly worse than *TopW* on IMDB and Yelp. For hard tasks (Figure 4.9), *TopW* has the lowest runtime for fast users while ALL is best on Mondial and FIRST on IMDB for slow users. From these results, we conclude that **in order to minimize the number of presented tuples, our algorithms incur some additional runtime overhead**. Consequently, **our algorithms are most beneficial for users who want to minimize tuple feedback** because they find it tedious or require much time to provide feedback on each tuple.

4.5 Related Work

Oblique query specification (OQS) OQS systems enable users to specify structured queries without requiring knowledge of the structured query language. Approaches include natural language interfaces [5,77] and query-by-example/query reverse engineering (QBE/QRE) [49] systems. The common thread among OQS approaches is that users provide imprecise query specifications which the system uses to generate candidate queries (CQs), and our goal is to help users select their target query from this set of candidate queries. Although our interaction model and QBE/QRE systems both use tuples as the main medium of interaction, they differ in that our model has the system suggest the tuples whereas users are the ones who provide tuples in QBE/QRE systems.

Target query selection Existing OQS systems enable users to whittle down the list of CQs in a *one-shot* or *iterative* fashion. The one-shot approach presents a full list of CQs to the user and asks them to select their target query, while the iterative approach allows the user to provide input which incrementally narrows the set of CQs to a final target query. While most existing systems opt for the one-shot approach [10, 61, 73], sample-driven schema mapping [59] and query from examples [44] are prominent examples of iterative interaction models. These existing approaches each suffer from at least one of the following failure modes:

- *Expecting user expertise.* [10, 73] output a list of ranked SQL queries and expect the user to select the correct one. This requires users to comprehend the database schema, defeating the very purpose of opting for the OQS interface in the first place. Query from examples [44] requires the user to examine query logic on synthetic data, which can be more challenging than labeling tuples from real data.
- *Failure to precisely distinguish CQs.* Some systems present alternate representations of CQs to aid users lacking SQL knowledge, such as natural language explanations [21,61]. However, these methods may provide identical summaries for two distinct CQs.

- *Wasted user effort.* For the sample-driven schema mapping model [59], user-suggested examples are not guaranteed to exist in the database instance, and even if they do, they may belong to the result sets of multiple CQs. As a result, very few CQs may be eliminated, and the user has wasted their time in coming up with and typing in such examples.

The distinguishing tuple model addresses each of these issues by presenting tuples as an easy-to-understand and precise means of distinguishing CQs, and by requiring the system to suggest tuples instead of the user.

Learning from membership queries The distinguishing tuple model is an application of the concept of learning with membership queries [4]. Previous work offers solutions that are tied to particular OQS workloads, such as learning join predicates [12, 13] and quantified queries [1]. In contrast, our method works with any OQS method, and also applies to settings where CQs have heterogeneous weights.

Interactive data exploration [22, 28] suggest interesting data to explore by enabling users to label system-suggested tuples. Their focus is on discovering interesting data patterns in the database with a set of tuples already known to the system, which involves a different series of optimization strategies from our setting where (possibly complex) candidate queries are provided and a target query must be selected from the CQ set while minimizing the number of CQ executions.

Decision trees Previous work [19, 40] in the area of decision trees seeks to distinguish a set of items using tests selected from a finite set. We apply a solution in the context of weighted items and uniform costs for tests [40] to our interaction model. We build on the general solution by also tackling the challenge of minimizing the cost of *generating* the set of tests (i.e. tuples).

4.6 Summary

In this chapter, we introduced the distinguishing tuple interaction model to tackle the target query selection problem. We formalized the problem of finding a minimal distinguishing tuple set, and proposed three algorithms to tackle this problem while limiting runtime. We demonstrated in evaluations that our algorithms could reduce user effort by up to 63% compared to the state-of-the-art. For future work, we hope to investigate the effects of user noise and integrate our approach seamlessly with existing OQS systems.

CHAPTER 5

Conclusion

In this dissertation, I argued that user domain expertise should be maximized to clarify OQS methods, so that users without technical expertise can reliably specify queries to relational databases. Existing OQS methods are deficient in that they are often unreliable when employed in isolation. Often a single specification is insufficient to triangulate the user's precise structured query, and consequently, it is ideal to solicit as much information as possible from the user, as multiple vectors of domain expertise can work in a complementary fashion to allow the system to converge at the user's desired query.

To this end, I presented a series of approaches to clarify OQS methods and evaluated them, answering the research questions in Table 1.3. I showed that we can effectively use information from previously-issued SQL queries on a database to guide existing OQS systems toward more likely user queries (Chapter 2), that we can design systems which can exploit the complementary effects of combining multiple specification methods (Chapter 3), and that we can assist the user in the process of target query selection by soliciting feedback on system-suggested tuples (Chapter 4).

The ultimate goal is to democratize data access by freeing non-technical users from needing to enlist the help of human technical staff in order to issue queries on a relational database. Accomplishing this goal could enable organizations to work more efficiently and removing technical overhead, and also could enable more of the general population to have access to insights from specific databases in a similar way to how the Internet and

search engines have opened access to knowledge previously buried in libraries and file cabinets.

Much future work remains with regard to this goal, and further work needs to be done to make these systems and technologies conventionally available.

5.1 Future Work

Some potential directions to extend the work in this dissertation are:

Exploring the bounds of domain expertise We investigated domain expertise in the context of "passive" knowledge embedded in SQL query logs and in the factual knowledge that takes the form of example tuples. There are several opportunities to explore different kinds of domain expertise. For example, what is the impact of experts' knowledge of domain terminology on querying databases? Are domain experts able to express logical constraints on what "realistic" data constitutes to help synthesize queries?

Building a monolithic multi-specification system We took a first step in Chapter 3 by combining two specification methods, natural language and programming-by-example. Further work needs to be done in extending this to a general purpose interface which is able to accept *any* specification the user desires, which truly maximizes the domain expertise of the user and reflects the way a user might interact with human technical support staff.

Streamlining iterative interaction When a query specification mode has failed to generate the user's desired query, the typical approach is to force the user to reformulate the specification. Interaction models which are explicitly designed for a human-in-the-loop, however, may be better able to reach the target query by allowing iterative refinement of the specification by the user.

Supporting semi-structured and unstructured data Much data in the world today resides outside of neatly structured relational databases. While we focused on the context of structured data in this dissertation, many of the ideas here can be extended to querying more unstructured data storage formats.

BIBLIOGRAPHY

[1] A. Abouzied, J. M. Hellerstein, and A. Silberschatz. Dataplay: interactive tweaking and example-driven correction of graphical database queries. In *The 25th Annual ACM Symposium on User Interface Software and Technology, UIST '12, Cambridge, MA, USA, October 7-10, 2012*, pages 207–218, 2012.

[2] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pages 5–16, 2002.

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

[4] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1987.

[5] 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.

[6] J. Berant, A. Chou, R. Frostig, and P. Liang. Semantic parsing on Freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, EMNLP 2013, 18-21 October 2013, Grand Hyatt Seattle, Seattle, Washington, USA, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1533–1544, 2013.

[7] S. Bergamaschi, E. Domnori, F. Guerra, R. T. Lado, and Y. Velegrakis. Keyword search over relational databases: a metadata approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 565–576, 2011.

[8] S. Bergamaschi, F. Guerra, M. Interlandi, R. T. Lado, and Y. Velegrakis. Combining user and database perspective for solving keyword queries over relational databases. *Inf. Syst.*, 55:1–19, 2016.

[9] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pages 431–440, 2002.

[10] L. Blunschi, C. Jossen, D. Kossmann, M. Mori, and K. Stockinger. SODA: generating SQL for business users. *PVLDB*, 5(10):932–943, 2012.

[11] 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.

[12] A. Bonifati, R. Ciucanu, and S. Staworko. Interactive inference of join queries. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014*, pages 451–462, 2014.

[13] A. Bonifati, R. Ciucanu, and S. Staworko. Learning join queries from user examples. *ACM Trans. Database Syst.*, 40(4):24:1–24:38, 2016.

[14] F. Brad, R. C. A. Iacob, I. Hosu, and T. Rebedea. Dataset for a neural natural language interface for databases (NNLIDB). In *Proceedings of the Eighth International Joint Conference on Natural Language Processing, IJCNLP 2017, Taipei, Taiwan, November 27 - December 1, 2017 - Volume 1: Long Papers*, pages 906–914, 2017.

[15] S. Brass and C. Goldberg. Semantic errors in SQL queries: A quite complete list. *Journal of Systems and Software*, 79(5):630–644, 2006.

[16] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.

[17] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini. Visual query systems for databases: A survey. *J. Vis. Lang. Comput.*, 8(2):215–260, 1997.

[18] E. Charniak. A maximum-entropy-inspired parser. In *6th Applied Natural Language Processing Conference, ANLP 2000, Seattle, Washington, USA, April 29 - May 4, 2000*, pages 132–139, 2000.

[19] F. Cicalese, T. Jacobs, E. S. Laber, and M. Molinaro. On greedy algorithms for decision trees. In *Algorithms and Computation - 21st International Symposium, ISAAC 2010, Jeju Island, Korea, December 15-17, 2010, Proceedings, Part II*, pages 206–217, 2010.

[20] M. de Marneffe, B. MacCartney, and C. D. Manning. Generating typed dependency parses from phrase structure parses. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation, LREC 2006, Genoa, Italy, May 22-28, 2006*, pages 449–454, 2006.

[21] D. Deutch, N. Frost, and A. Gilad. Provenance for natural language queries. *PVLDB*, 10(5):577–588, 2017.

[22] K. Dimitriadou, O. Papaemmanouil, and Y. Diao. AIDE: an active learning-based approach for interactive data exploration. *IEEE Trans. Knowl. Data Eng.*, 28(11):2842–2856, 2016.

[23] L. Dong and M. Lapata. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*, 2016.

[24] M. Eirinaki, S. Abraham, N. Polyzotis, and N. Shaikh. Querie: Collaborative database exploration. *IEEE Trans. Knowl. Data Eng.*, 26(7):1778–1790, 2014.

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

[26] FriendlyData. FriendlyData. https://friendlydata.io/. Accessed: 2018-03-28.

[27] R. Ge and R. J. Mooney. A statistical semantic parser that integrates syntax and semantics. In *Proceedings of the Ninth Conference on Computational Natural Language Learning, CoNLL 2005, Ann Arbor, Michigan, USA, June 29-30, 2005*, pages 9–16, 2005.

[28] X. Ge, Y. Xue, Z. Luo, M. A. Sharaf, and P. K. Chrysanthis. REQUEST: A scalable framework for interactive construction of exploratory queries. In *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*, pages 646–655, 2016.

[29] W. H. Gomaa and A. A. Fahmy. A survey of text similarity approaches. *International Journal of Computer Applications*, 68(13):13–18, April 2013.

[30] 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.

[31] M. Hart and M. Blythe. Q&A in Power BI service and Power BI Desktop. https://docs.microsoft.com/en-us/power-bi/power-bi-q-and-a. Accessed: 2018-03-28.

[32] K. Hashimoto, C. Xiong, Y. Tsuruoka, and R. Socher. A joint many-task model: Growing a neural network for multiple NLP tasks. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, pages 1923–1933, 2017.

[33] V. Hristidis and Y. Papakonstantinou. DISCOVER: keyword search in relational databases. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*, pages 670–681, 2002.

[34] S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer. Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 963–973, 2017.

[35] Z. Jin, C. Baik, M. J. Cafarella, and H. V. Jagadish. Beaver: Towards a declarative schema mapping. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, pages 10:1–10:4, 2018.

[36] Z. Jin, C. Baik, M. J. Cafarella, H. V. Jagadish, and Y. Lou. Demonstration of a multiresolution schema mapping system. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[37] R. M. Karp. Reducibility among combinatorial problems. In *Proceedings of a Symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, pages 85–103, 1972.

[38] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. Snipsuggest: Context-aware autocompletion for SQL. *PVLDB*, 4(1):22–33, 2010.

[39] D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics, 7-12 July 2003, Sapporo Convention Center, Sapporo, Japan*, pages 423–430, 2003.

[40] S. R. Kosaraju, T. M. Przytycka, and R. S. Borgstrom. On an optimal split tree problem. In *Algorithms and Data Structures, 6th International Workshop, WADS '99, Vancouver, British Columbia, Canada, August 11-14, 1999, Proceedings*, pages 157–168, 1999.

[41] L. T. Kou, G. Markowsky, and L. Berman. A fast algorithm for Steiner trees. *Acta Informatica*, 15:141–145, 1981.

[42] Y. Lee, K. A. Kozar, and K. R. T. Larsen. The technology acceptance model: Past, present, and future. *CAIS*, 12:50, 2003.

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

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

[45] Y. Li and D. Rafiei. Natural language data management and interfaces: Recent development and open challenges. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1765–1770, 2017.

[46] Y. Li, H. Yang, and H. V. Jagadish. NaLIX: an interactive natural language interface for querying XML. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 900–902, 2005.

[47] P. Liang. Learning executable semantic parsers for natural language understanding. *Commun. ACM*, 59(9):68–76, 2016.

[48] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, June 22-27, 2014, Baltimore, MD, USA, System Demonstrations*, pages 55–60, 2014.

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

[50] W. May. Information extraction and integration with Florid: The Mondial case study. Technical Report 131, Universität Freiburg, Institut für Informatik, 1999. Available from http://dbis.informatik.uni-goettingen.de/Mondial.

[51] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, pages 3111–3119, 2013.

[52] G. A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, 1995.

[53] A. Nandi and H. V. Jagadish. Qunits: queried units in database search. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*, 2009.

[54] J. Nivre, M. de Marneffe, F. Ginter, Y. Goldberg, J. Hajic, C. D. Manning, R. T. McDonald, S. Petrov, S. Pyysalo, N. Silveira, R. Tsarfaty, and D. Zeman. Universal dependencies v1: A multilingual treebank collection. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation LREC 2016, Portorož, Slovenia, May 23-28, 2016*, 2016.

[55] 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.

[56] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543, 2014.

[57] 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.

[58] 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.

[59] 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.

[60] T. Russell-Rose and T. Tate. Chapter 1 - the user. In T. Russell-Rose and T. Tate, editors, *Designing the Search Experience*, pages 3 – 21. Morgan Kaufmann, 2013.

[61] 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.

[62] S. Schuster and C. D. Manning. Enhanced english universal dependencies: An improved representation for natural language understanding tasks. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation LREC 2016, Portorož, Slovenia, May 23-28, 2016*, 2016.

[63] 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.

[64] solid IT gmbh. DBMS popularity broken down by database model. https://db-engines.com/en/ranking_categories. Accessed: 2019-04-01.

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

[66] L. R. Tang and R. J. Mooney. Using multiple clause constructors in inductive logic programming for semantic parsing. In *Machine Learning: EMCL 2001, 12th European Conference on Machine Learning, Freiburg, Germany, September 5-7, 2001, Proceedings*, pages 466–477, 2001.

[67] S. Tata and G. M. Lohman. SQAK: doing more with keywords. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 889–902, 2008.

[68] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42, 2006.

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

[70] C. Unger, L. Bühmann, J. Lehmann, A. N. Ngomo, D. Gerber, and P. Cimiano. Template-based question answering over RDF data. In *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012, Lyon, France, April 16-20, 2012*, pages 639–648, 2012.

[71] P. Utama, N. Weir, F. Basik, C. Binnig, U. Çetintemel, B. Hättasch, A. Ilkhechi, S. Ramaswamy, and A. Usta. An end-to-end neural natural language interface for databases. *CoRR*, abs/1804.00401, 2018.

[72] C. Van Rijsbergen, S. Robertson, and M. Porter. *New Models in Probabilistic Information Retrieval.* British Library Research & Development Report. Computer Laboratory, University of Cambridge, 1980.

[73] 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.

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

[75] 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.

[76] X. Xu, C. Liu, and D. Song. SQLNet: generating structured queries from natural language without reinforcement learning. *CoRR*, abs/1711.04436, 2017.

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

[78] P. Yin, Z. Lu, H. Li, and B. Kao. Neural enquirer: Learning to query tables in natural language. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 2308–2314, 2016.

[79] T. Yu, M. Yasunaga, K. Yang, R. Zhang, D. Wang, Z. Li, and D. R. Radev. SyntaxSQL-Net: 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.

[80] 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.

[81] V. Zhong, C. Xiong, and R. Socher. Seq2SQL: generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017.

[82] 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.