Querying RDBMS Using Natural Language

by

Fei Li

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
2017

Doctoral Committee:

Professor Hosagrahar V. Jagadish, Chair
Professor Dragomir R. Radev
Associate Professor Michael J. Cafarella
Associate Professor Qiaozhu Mei

Fei Li

lifei@umich.edu

ORCID iD: 0000-0002-8731-6774

Acknowledgments

I am forever in debt to my advisor, Prof. H. V. Jagadish for making this dissertation even possible. His belief in me often far exceeded my belief in myself. As my intellectual father and mentor, he provided me with all the supports that I needed to complete this journey when I was in doubt. The support was not solely sharp and intellectual, but also warm and emotional one that carried me throughout my career in Ann Arbor. Without him, I would not have been able to accomplish what I have done and live in the way I want today.

I would also like to sincerely thank my dissertation committee, Professor Dragomir Radev, Professor Michael J. Cafarella and Professor Qiaozhu Mei, for spending their valuable time on my dissertation. Their insightful comments and guidance have greatly improved the quality of my dissertation.

It has been an honor to be part of the University of Michigan Database Group. I would like to thank former and current colleagues Anna Shaverdian, Bin Liu, Chun-Hung Hsiao, Dan Fabbri, Dolan Antenucci, Fernando Farfan, Glenn Tarcea, Jing Zhang, Jie Song, Li Qian, Lujun Fang, Manish Singh, Matthew Burgess, Michael Anderson, Nikita Bhutani, Yongjoo Park, Zhe Chen and Zhongjun Jin, for all the support and joy you have brought to me over these years.

And above all, I would like to thank my wife. Thank you for your love, care, understanding, support and blessings.

TABLE OF CONTENTS

Acknowledgments . . . . . **ii**

List of Figures . . . . . **vi**

Abstract . . . . . **viii**

Chapter

1 Introduction **1**

1.1 Why NLIDB? 1
1.2 Challenges & Strategies 3
1.2.1 Interactive Communications 3
1.2.2 Semantic Coverage as weighted SQL Templates 4
1.2.3 Managing the Variants of NLQ Expressions 4
1.3 Key Contributions & Dissertation Outline 5

2 Constructing an Interactive Interface for Relational Databases **7**

2.1 Introduction 7
2.2 Query Mechanism 9
2.3 System Overview 12
2.4 Parse Tree Node Interpretation 14
2.4.1 Data Model 15
2.4.2 Candidate Mappings 16
2.4.3 Default Mapping Strategy 17
2.5 Parse Tree Structure Adjustment 18
2.5.1 Query Tree 19
2.5.2 Parse Tree Reformulation 20
2.5.3 Implicit Nodes Insertion 22
2.6 SQL Generation 24
2.6.1 Basic Translation 25
2.6.2 Blocks and Subqueries 25
2.7 Experiments 27
2.7.1 User Interface 27
2.7.2 Measurement 28
2.7.3 Experiments Design 28
2.7.4 Results and Analysis 31
2.8 Summary 33

3 A Template-based Interface for Relational Databases . . . 34

3.1 Introduction . . . 34
3.2 Overview . . . 38
3.2.1 SQL Template . . . 38
3.2.2 Problem Definition . . . 40
3.2.3 System Architecture . . . 41
3.3 Template Generation . . . 43
3.4 Popularity . . . 45
3.4.1 Template Relevance . . . 46
3.4.2 Popularity of SQL Templates . . . 47
3.5 Online Query Interpretation . . . 49
3.5.1 Element Mapping and Tokenization . . . 49
3.5.2 Template Mapping . . . 51
3.6 Experiments . . . 52
3.6.1 Offline Part . . . 52
3.6.2 Online Part . . . 54
3.7 Summary . . . 57

4 Interpreting Natural Language Queries over Databases . . . 58

4.1 Introduction . . . 58
4.2 Preliminaries . . . 62
4.2.1 Semantic Coverage . . . 62
4.2.2 SQL Template . . . 63
4.3 Query Mapping . . . 64
4.3.1 Information Overlap . . . 65
4.3.2 Semantic Distance between Queries . . . 66
4.3.3 Template Mapping . . . 68
4.4 Entity Disambiguation . . . 69
4.5 System Architecture . . . 72
4.6 Experiments . . . 73
4.6.1 Disambiguation at Entity Level . . . 74
4.6.2 Template Mapping . . . 76
4.7 Conclusion . . . 78

5 Related Work . . . 79

5.1 Semantic Parsing . . . 79
5.2 Generic NLIDB . . . 80
5.3 Question Answer Systems on Knowledge Base . . . 80
5.4 Keyword Search over Relational Databases . . . 81
5.5 Other Kinds of User Friendly Interfaces for Databases . . . 82
5.5.1 Form-based Search . . . 82
5.5.2 Faceted Search . . . 82
5.5.3 Visual Query Builders . . . 82
5.5.4 Query by Examples . . . 83
5.5.5 Schema Free SQL/XQuery . . . 83

5.5.6 Schema Summarization 83
5.6 Search Engines 84
5.7 Word-embedding & Sentence Embedding 84
5.8 Entity Matching 85
5.9 Query Explanation 85
6 Conclusions and Future Work **86**
Bibliography **88**

LIST OF FIGURES

1.1 An Overview of Different Query Methods. 2

2.1 A Simplified Schema for Microsoft Academic Search and Sample Queries. . . 8
2.2 System Architecture for NaLIR. 10
2.3 (a) A Simplified Linguistic Parse Tree from the Stanford Parser. (b) A Mapping Strategy for the Nodes in the Parse Tree. (c) A Valid Parse Tree. (d) A Query Tree after Inserting Implicit Nodes. 14
2.4 Different Types of Nodes in the Tokenization. 15
2.5 Grammar of Valid Parse Trees. 19
2.6 Parse Tree Reformulation Algorithm. 21
2.7 (a) A Simplified Linguistic Parse Tree for Query 3 in Figure 2.1. (b) A Valid Parse Tree. (c) A Query Tree after Inserting Implicit Nodes. 22
2.8 Translated SQL Statement for the Query Tree in Figure 2.7 (c). 26
2.9 Query Interface of NaLIR 29
2.10 Statistics for MAS Database. 30
2.11 Sample Queries in Different Complexity. 30
2.12 Effectiveness. 32
2.13 Failures in each Component. 32
2.14 Average Time Cost (s). 33

3.1 Simplified Schema Graph for Microsoft Academic Search 35
3.2 Sample SQL Template. 38
3.3 System Architecture for TBNaLIR. 39
3.4 Pairs of Relevant Queries. 46
3.5 Types of Nodes. 49
3.6 Statistics for MAS. 52
3.7 Quality of the Semantic Coverage. 54
3.8 Experimental Results for the Online Mapping. 56

4.1 Schema Graph for MAS 59
4.2 A Sample SQL Template. 64
4.3 Word to Vector Embedding. 67
4.4 Different Types of Phrases. 69
4.5 Number of Subgraphs by each Entity Combination 71
4.6 System Architecture. 71
4.7 Summary Statistics for MAS Dataset. 74

4.8 Samples of Groups of Entities. 75
4.9 Entity Disambiguation. 76
4.10 Results for Template Mapping. 78

ABSTRACT

It is often challenging to specify queries against a relational database since SQL requires its users to know the exact schema of the database, the roles of various entities in a query, and the precise join paths to be followed. On the other hand, keyword search is unable to express many desired query semantics.

In the real world, people ask questions in natural language, such as English. Theoretically, natural language interfaces for databases (NLIDBs) have many advantages over other widely accepted query interfaces (keyword-based search, form-based interface, and visual query builder). For example, a typical NLIDB would enable naive users to specify complex, ad-hoc query intent without training. Not surprisingly, an NLIDB is regarded by many as the ultimate goal. Despite these advantages, in real world applications, NLIDBs have not been widely adopted.

In this dissertation, we investigate the construction of NLIDBs, specifically from the following three aspects:

1. A natural language query is inherently ambiguous and some ambiguities may be too hard for computers to resolve. Can a system collaborate with users to achieve satisfactory reliability without burdening the user too much?

2. The interpretation process can be considered as a mapping from a natural language query to the correct point in the semantic coverage of the NLIDB. Can the mapping process get easier by carefully defining the semantic coverage?

3. Can an NLIDB work when no training examples are available, collect the user behavior data as the training examples and improve itself from real usage?

In this dissertation, we provide affirmative answers to the above questions in the form of new query mechanism designed, techniques provided and systems constructed.

CHAPTER 1

Introduction

This dissertation studies the problem of building natural language interfaces to databases (NLIDBs), with a focus on relational databases. In this chapter, we begin with the motivation to study NLIDBs by presenting a short comparison of NLIDB with other kinds of query systems. We follow with some of our observations of the challenges in constructing NLIDBs and provide our general strategies. Finally, we conclude with the novelties and an outline on the work presented in the rest of the dissertation.

1.1 Why NLIDB?

Querying data in relational databases is often challenging. SQL is the standard query language for relational databases. While expressive and powerful, SQL is too difficult for users without technical training. Even for users with expertise in database programming languages, it can be challenging because it requires that users know the exact schema of the database, the roles of various entities in a query, and the precise join paths to be followed. This difficulty is exacerbated by normalization, a process that is central to relational database design, which brings benefits including saved space and avoided update anomalies, at the cost of spreading data across several relations and thus making the database schema more complex. As the database user base is shifting towards non-experts, designing user-friendly query interfaces will be a more important goal in database community.

Designing an interface for naive users[1] to query a database is not an easy task. First, users may tend to express complex query semantics when interacting with databases since they know databases are more than collections of documents and there are structures inside databases that can be used to answer their queries. Second, users expect precise and complete answers from database queries, which means anything less than perfect precision and recall

[1] In this dissertation, the only characteristic for naive users is that they are not familiar with programming languages. Note that naive users may be scientists in other area who are in need to query complex scientific databases.

Standard Query System: Semantic meaning is perfectly defined

Methods	Support Complex Query Logic	Support Ad-hoc Query Logic	Can be used without Training
SQL	✔	✔	
Visual Query Builder	✔	✔	
Form-based Search	✔		✔

Heuristic Query Systems: The goal is to infer the user intent

Methods	Support Complex Query Logic	Support Ad-hoc Query Logic	Can be used without Training
Keyword Search		✔	✔
NLIDB	✔	✔	✔

Figure 1.1: An Overview of Different Query Methods.

will have to be explained to the user. These needs make designing such an interface, or even just a query mechanism, hard.

In the real world, people ask questions in natural language, such as English. Not surprisingly, a natural language interface is regarded by many as the ultimate goal for a database query interface, and many NLIDBs have been built towards this goal [5, 69, 51, 61, 47, 26, 57, 54, 70]. NLIDBs have many advantages over other widely accepted query interfaces (keyword-based search, form-based interface, and visual query builder). For example, a typical NLIDB would enable naive users to specify *complex*, *ad-hoc* query intent *without training*. In contrast, flat-structured keywords are often insufficient to convey complex query intent, form-based interfaces can be used only when queries are predictable and limited to the encoded logic, and visual query builders still requires extensive schema knowledge of the user. A summarization of different query mechanisms is shown in Figure 1.1[2].

[2]In this dissertation, we divide query systems into two categories: (a) *standard query systems*, in which the semantic meaning of each query is formally defined, and (b) *heuristic query systems*, in which queries have ambiguities and the goal for the system is to infer the user intent. Obviously, NLIDB is in the second category.

1.2 Challenges & Strategies

Despite these advantages, NLIDBs have not been adopted widely in real world applications. In the remaining parts of this section, we discuss some of our observations for the possible challenges and provide our general strategies.

1.2.1 Interactive Communications

NLIDBs normally adopt the query mechanism of one answer for one question, with limited backup answers or explanations. But an NLIDB is a heuristic query system, in which its queries do not have formally defined semantic meanings. The goal for an NLIDB is to infer the user intent and this task is regarded by many as an "AI complete problem". Therefore, we cannot reasonably expect an NLIDB to be perfect. Users may be provided with a wrong answer due to the system incorrectly understanding or handling the query. The system does not help users detect such error and sometimes it is impossible for users to verify the answer by themselves. Even if the user does realize that the answer is wrong, there is little guidance on what to do. The only option available to the user is to rephrase the query and hope that the system now understands it better.

We observe that when humans communicate with one another in natural language, the query-response cycle is not as rigid as in a traditional database system. If a human is asked a query that she does not understand, she will seek clarification. She may do so by asking specific questions back, so that the question-asker understands the point of potential confusion. She may also do so by stating explicitly how she interpreted the query.

Drawing inspiration from this natural human behavior, we design the query mechanism to facilitate collaboration between the system and the user in processing natural language queries. First, the system explains how it interprets a query, from each ambiguous word/phrase to the meaning of the whole sentence. These explanations enable the user to verify the answer and to be aware where the system misinterprets her query. Second, for each ambiguous part, we provide multiple likely interpretations for the user to choose from. Since it is often easier for users to recognize an expression rather than to compose it, we believe this query mechanism can achieve satisfactory reliability without burdening the user too much. To support such a query mechanism, we design a query tree structure to represent the interpretation of a natural language query from the databases perspective. A query tree can be explained to the user for verification, and once verified, will almost always be correctly translated to SQL. We also provide a modular architecture to support such a query mechanism, in which each component can be designed and improved independently.

1.2.2 Semantic Coverage as weighted SQL Templates

When using NLIDBs, naive users tend to use informal expressions and prefer brevity to grammatical correctness, which makes the query interpretation harder. We find that humans can often easily understand an ambiguous natural language query, even in the cases when it is severely underspecified. We believe that the main reason is that humans have an intuitive understanding of which interpretation is more reasonable than others. Specifically, common sense, previous experiences, and domain knowledge serve as resources for our brains to develop such intuitions. We find that in database applications, query log, underlying schema structure and data distribution can serve as valuable resources to obtain "similar intuitions". These resources make the problem of constructing an NLIDB different from its general problem of building a semantic parser. We explore the possibility in configuring NLIDBs with such intuitions, in a generic manner independent of domains, to resolve the ambiguities in natural language queries.

Specifically, we model the semantic coverage of an NLIDB as a set of weighted SQL templates, in which the weight describes the likelihood of each template to be queried. Such set of weighted SQL templates are generated by analyzing all kinds of resources that are available. Then the problem of interpreting an natural language query is modeled as a mapping problem, from the natural language query to the SQL templates in the semantic coverage. In the mapping, the SQL templates with high weights are preferred. By taking the weights into account, the precision and recall of the mapping are fundamentally enhanced. We further propose a framework for NLIDBs comprising two main parts: an offline part, which is responsible for generating the weighted semantic coverage, and an online part, which is responsible for mapping online natural language queries to the query logics.

1.2.3 Managing the Variants of NLQ Expressions

Given the rich varieties of natural language expressions and the huge gap between a natural language query and its corresponding SQL query, many natural language queries are hard to interpret correctly based only on schema mapping. We observe that although the potential number of different expressions might be large, the expressions that are likely to be frequently used by real users are often in limited numbers of groups, in which the expression in each group are only different in stop words, word order or synonyms. That means, for a query logic with a few different natural language expressions previously recorded, each new natural language expression describing the same query logic should be, in most cases, very similar to at least one of the previous expressions.

As such, in our system, we would like to benefit from training examples to manage the

variants of the natural language expressions. Specifically, we present metrics to detect the similarity between natural language expressions. The metrics we provided are hyperparameter free, which means they can be used even in the cases when the training examples are very limited. At the cold start stage, we use generic metrics without hyperparameters to evaluate the similarity between an NLQ and a SQL template.

To collect the training data, we adopt the interactive communications in our first work. Top mapped SQL templates with value instantiations are explained for the user to choose from. Once the user make a choice, NLQ paired with the chosen SQL template is collected, which serves as the prefect training set to improve the system.

1.3 Key Contributions & Dissertation Outline

The dissertation studies the challenges in constructing NLIDBs. To ensure *reliability*, our system explains to the user how her query is interpreted step by step for verification. The whole process is in natural language to make it understandable for naive users. When ambiguities exist, our system will return the user multiple interpretations with explanations for the user to choose from, which solves the ambiguities *interactively* with the user. Each time when a user makes a choice and confirm the interpretation, a *training example* is collected from the usage.

We design a template-based NLIDB, which models the semantic coverage of an NLIDB as a set of weighted SQL templates. The weight of a SQL template indicates whether the query logic is *semantically meaningful* and likely to be frequently queried, which provides important evidence for solving ambiguities. The semantic coverage is generated by analyzing different kinds of *resources* like the SQL query log, specifications from domain experts, schema structure and the data distribution of the database, which can provide evidence for solving ambiguities but are not often investigated in previous literatures.

With the semantic coverage that are carefully defined, the interpretation of a natural language is simplified to a *mapping problem*, from the natural language query to the desired SQL template in the semantic coverage. By this design, even in the cases when some mistakes exist in the intermediate results like wrong linguistic parse tree generated, the system still has a large chance to map it to the correct SQL template. In the *cold start* stage when no training examples are available, the mapping is mainly based on schema mapping techniques that can be used across different domains. Though the collection of training examples in usage, the system utilizes the training examples to improve the ranking. The learning strategy is hyperparameter free, which are very suitable for the cases when training examples are limited.

The rest of the dissertation is organized as follows. Chapter 2 discusses the construction of an interactive NLIDB. In Chapter 3, we define the semantic coverage of an NLIDB and provide a template-based solution for interpreting natural language queries. In Chapter 4, we propose the strategy to improve the system from the training examples collected. Chapter 5 elaborates on related work. Finally, Chapter 6 concludes the dissertation.

CHAPTER 2

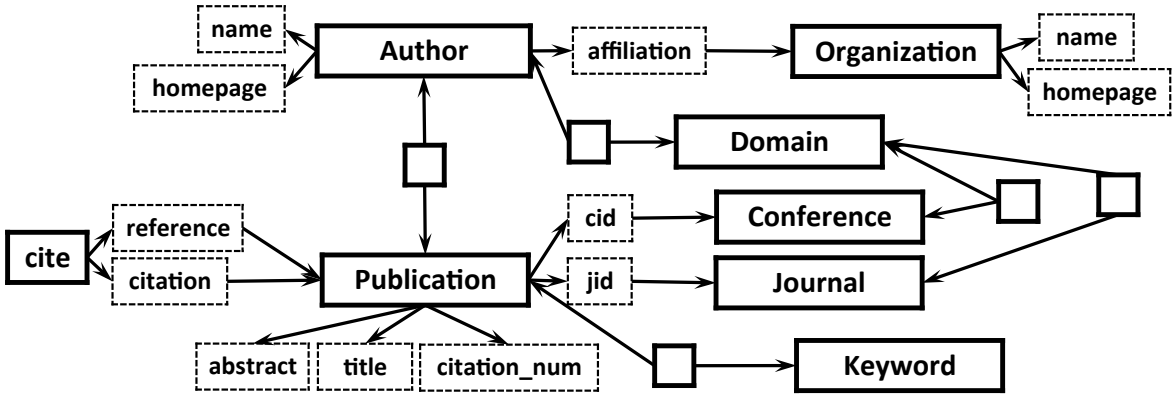
Constructing an Interactive Interface for Relational Databases

2.1 Introduction

Traditionally, research work in querying data from relational databases often follows one of two paths: the structured query approach and the keyword-based approach. Both approaches have their advantages and disadvantages. The structured query approach, while expressive and powerful, is not easy for naive users. The keyword-based approach is very friendly to use, but cannot express complex query intent accurately. In contrast, natural language has both advantages to a large extent: even naive users are able to express complex query intent in natural language. Thus supporting natural language queries is often regarded as the ultimate goal for a database query interface.

However, progress has been slow, even as general natural language processing systems have improved over the years. The fundamental problem is that understanding natural language is hard. People may use slang words, technical terms, and dialect-specific phrasing, none of which may be known to the system. Even without these, natural language is inherently ambiguous. Even in human-to-human interaction, there are miscommunications. Therefore, we cannot reasonably expect an NLIDB to be perfect. Therefore, users may be provided with a wrong answer due to the system incorrectly understanding or handling the query. The system does not help user detect such error and sometimes it is impossible for users to verify the answer by themselves. So a user cannot be sure that the answer provided is really the answer to the question asked. Moreover, even if the user does realize that the answer is wrong, there is little guidance on what to do. The only option available to the user is to rephrase the query and hope that the system now understands it better.

When humans communicate with one another in natural language, the query-response cycle is not as rigid as in a traditional database system. If a human is asked a query that



Query 1: Return the average number of publications by Bob in each year.
Query 2: Return authors who have more papers than Bob in VLDB after 2000.
Query 3: Return the conference in each area whose papers have the most total citations.

Figure 2.1: A Simplified Schema for Microsoft Academic Search and Sample Queries.

she does not understand, she will seek clarification. She may do so by asking specific questions back, so that the question-asker understands the point of potential confusion. She may also do so by stating explicitly how she interpreted the query. Drawing inspiration from this natural human behavior, we design the query mechanism to facilitate collaboration between the system and the user in processing natural language queries. First, the system explains how it interprets a query, from each ambiguous word/phrase to the meaning of the whole sentence. These explanations enable the user to verify the answer and to be aware where the system misinterprets her query. Second, for each ambiguous part, we provide multiple likely interpretations for the user to choose from. Since it is often easier for users to recognize an expression rather than to compose it, we believe this query mechanism can achieve satisfactory reliability without burdening the user too much.

A question that then arises is how should a system represent and communicate its query interpretation to the user. SQL is too difficult for most non-technical humans. We need a representation that is both "human understandable" and "RDBMS understandable". In this chapter, we present a data structure, called *Query Tree*, to meet this goal. As an intermediate between a linguistic parse tree and a SQL statement, a query tree is easier to explain to the user than a SQL statement. Also, given a query tree verified by the user, the system will almost always be able to translate it into a correct SQL statement.

Putting the above ideas together, we propose an NLIDB comprising three main components: a first component that transforms a natural language query to a query tree, a second component that verifies the transformation interactively with the user, and a third component that translates the query tree into a SQL statement. We have constructed such an NLIDB, and we call it a NaLIR (Natural Language Interface to Relational databases).

The intellectual contributions of this chapter are as follows:

1. *Interactive Query Mechanism.* We design an interaction mechanism for NLIDBs to enable users to ask complex queries and have them interpreted correctly, with a little interaction help.

2. *Query Tree.* We design a query tree structure to represent the interpretation of a natural language query from the database's perspective. A query tree can be explained to the user for verification, and once verified, will almost always be correctly translated to SQL.

3. *System Architecture.* We provide a modular architecture to support such a query mechanism, in which each component can be designed, and improved, independently. We develop a working software system called NaLIR, instantiating this architecture. We discuss the basic ideas in designing heuristic functions for each component and describe the specific choices made in our system.

4. *User Study.* We demonstrate, through carefully designed user studies, that NaLIR is usable in practice, on which even naive users are able to handle quite complex ad-hoc query tasks.

The remaining parts of the paper are organized as follows. We discuss the query mechanism in Section 2.2. The system architecture of our system is described in Section 2.3. Given a query, we show how to interpret each its words/phrases in Section 2.4 and infer the semantic meaning of the whole query (represented by a query tree) in Section 2.5. We discuss how our system translates a query tree to a SQL statement in Section 2.6. In Section 2.7, our system is evaluated experimentally. In Section 2.8, we summary the chapter.

2.2 Query Mechanism

Keyword search systems are popular and effective in at least some domains, such as for document search. As we think about the architecture of an NLIDB, it is worthwhile to draw inspiration from search engines, and how they infer user intent from limited information. First, given a query, a search engine returns a list of results, rather than a single result. This is central to providing acceptable recall. Second, users are able to verify whether a result is correct (useful) by reading the abstract/content. Third, these results are well ranked, to minimize user burden to verify potential answers. These strategies work very well in search engines. However, due to some fundamental differences between search engines

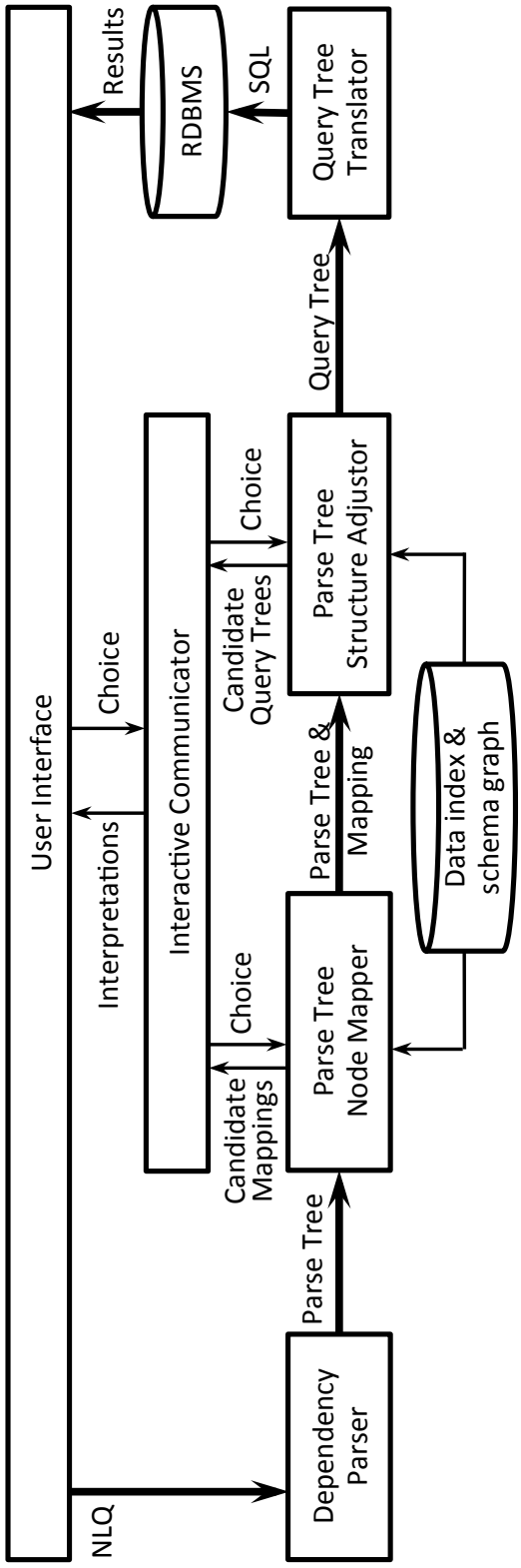


Figure 2.2: System Architecture for NaLIR.

and NLIDBs, as we will discuss next, this query mechanism cannot be directly applied to NLIDBs.

First, users are often able to verify the results from a search engine by just reading the results. However, the results returned by an NLIDB cannot usually explain themselves. For example, suppose a user submits Query 1 in Figure 2.1 to an NLIDB and gets an answer "5". How could she verify whether the system understands her query and returns to her the correct answer? To facilitate this, an NLIDB should provide explanations for the result at least in terms of how the query is processed.

Second, unlike search engines, users tend to express *sophisticated query logics* to an NLIDB and expect *perfect results*. That requires the NLIDB to fix all the ambiguities and get a perfect interpretation for the query from the perspective of both linguistics and the given database. However, natural language queries are often inherently ambiguous and sometimes, some of the ambiguities are too "hard" for systems to fix with confidence. Consider Query 1 in Figure 2.1 again, the user specifies the word "publication", while in a real world bibliography database, the information of publications may be stored in many tables, say, article, book, incollection, phdThesis, journal, proceedings and so forth. The system itself cannot be expected to figure out which ones of these should be considered as publications. Even if the included items are all unambiguous, when natural language queries have complex structures with modifier attachments, aggregations, comparisons, quantifiers, and conjunctions, it may contain several ambiguities that cannot be fixed by the system with confidence, from the interpretation of an ambiguous phrase to the relationship between words/phrases. The number of possible interpretations grows exponentially with the number of unfixed ambiguities. As a result, there may be hundreds of candidate interpretations for a complex natural language query, of which only one is correct. Since these interpretations are often similar to each other in semantics, it is very hard to develop an effective ranking function for them. As a result, if the system simply returns a list of hundreds of answers, the users will be frustrated in verifying them.

Given the above two observations, instead of explaining the query results, we explain the query interpretation process, especially how each ambiguity is fixed, to the user. In our system, we fix each "easy" ambiguity quietly. For each "hard" ambiguity, we provide multiple interpretations for the user to choose from. In such a way, even for a rather complex natural language query, verifications for 3-4 ambiguities is enough, in which each verification is just making choices from several options.

The ambiguities in processing a natural language query are not often independent of each other. The resolution of some ambiguities depends on the resolution of some other ambiguities. For example, the interpretation of the whole sentence depends on how each of

its words/phrases is interpreted. So the disambiguation process and the verification process should be organized in a few steps. In our system, we organize them in three steps, as we will discuss in detail in the next section. In each step, for a "hard" ambiguity, we generate multiple interpretations for it and, at the same time, use the best interpretation as the default choice to process later steps. Each time a user changes a choice, our system immediately reprocesses all the ambiguities in later steps and updates the query results.

2.3 System Overview

Figure 2.2 depicts the architecture of NaLIR[1]. The entire system we have implemented consists of three main parts: the query interpretation part, interactive communicator and query tree translator. The query interpretation part, which includes *parse tree node mapper* (Section 2.4) and *structure adjustor* (Section 2.5), is responsible for interpreting the natural language query and representing the interpretation as a query tree. The *interactive communicator* is responsible for communicating with the user to ensure that the interpretation process is correct. The query tree, possibly verified by the user, will be translated into a SQL statement in the *query tree translator* (Section 2.6) and then evaluated against an RDBMS.

Dependency Parser. The first obstacle in translating a natural language query into a SQL query is to understand the natural language query linguistically. In our system, we use the Stanford Parser [24] to generate a linguistic parse tree from the natural language query. The linguistic parse trees in our system are dependency parse trees, in which each node is a word/phrase specified by the user while each edge is a linguistic dependency relationship between two words/phrases. The simplified linguistic parse tree of Query 2 in Figure 2.1 is shown in Figure 2.3 (a).

Parse Tree Node Mapper. The parse tree node mapper identifies the nodes in the linguistic parse tree that can be mapped to SQL components and tokenizes them into different tokens. In the mapping process, some nodes may fail in mapping to any SQL component. In this case, our system generates a warning to the user, telling her that these nodes do not directly contribute in interpreting her query. Also, some nodes may have multiple mappings, which causes ambiguities in interpreting these nodes. For each such node, the parse tree node mapper outputs the best mapping to the parse tree structure adjustor by default and reports all candidate mappings to the interactive communicator.

Parse Tree Structure Adjustor. After the node mapping (possibly with interactive communications with the user), we assume that each node is understood by our system. The next

[1]In the current implementation, we use MySQL as the RDBMS, and Stanford NLP Parser [24] as the dependency natural language parser.

step is to correctly understand the tree structure from the database's perspective. However, this is not easy since the linguistic parse tree might be incorrect, out of the semantic coverage of our system or ambiguous from the database's perspective. In those cases, we adjust the structure of the linguistic parse tree and generate candidate interpretations (query trees) for it. In particular, we adjust the structure of the parse tree in two steps. In the first step, we reformulate the nodes in the parse tree to make it fall in the syntactic coverage of our system (valid parse tree). If there are multiple candidate valid parse trees for the query, we choose the best one as default input for the second step and report top k of them to the interactive communicator. In the second step, the chosen (or default) valid parse tree is analyzed semantically and implicit nodes are inserted to make it more semantically reasonable. This process is also under the supervision of the user. After inserting implicit nodes, we obtain the exact interpretation, represented as a query tree, for the query.

Interactive Communicator. In case the system possibly misunderstands the user, the interactive communicator explains how her query is processed. In our system, interactive communications are organized in three steps, which verify the intermediate results in the parse tree node mapping, parse tree structure reformulation, and implicit node insertion, respectively. For each ambiguous part, we generate a multiple choice selection panel, in which each choice corresponds to a different interpretation. Each time a user changes a choice, our system immediately reprocesses all the ambiguities in later steps.

Example 1 *Consider the linguistic parse tree T in Figure 2.3(a). In the first step, the parse tree node mapper generates the best mapping for each node (represented as M and shown in Figure 2.3 (b)) and reports to the user that the node "VLDB" maps to "VLDB conference" and "VLDB Journal" in the database and that our system has chosen "VLDB conference" as the default mapping. According to M, in the second step, the parse tree adjustor reformulates the structure of T and generates the top k valid parse trees $\{T_i^M\}$, in which T_1^M (Figure 2.3 (c)) is the best. The interactive communicator explains each of the k valid parse trees in natural language for the user to choose from. For example, T_1^M is explained as "return the authors, where the papers of the author in VLDB after 2000 is more than Bob". In the third step, T_1^M is fully instantiated in the parse tree structure adjustor by inserting implicit nodes (shown in Figure 2.3 (d)). The result query tree T_{11}^M is explained to the user as "return the authors, where the number of papers of the author in VLDB after 2000 is more than the number of paper of Bob in VLDB after 2000.", in which the underline part can be canceled by the user. When the user changes the mapping strategy M to M', our system will immediately use M' to reprocess the second and third steps. Similarly, if the user choose T_i^M instead of T_1^M as the best valid parse tree, our system will fully instantiate T_i^M in the third step and update the interactions.*

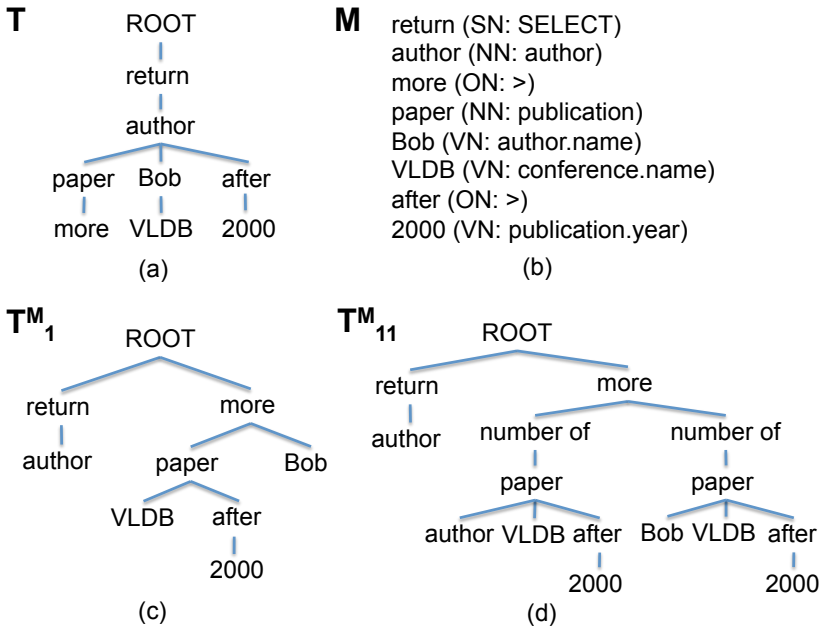


Figure 2.3: (a) A Simplified Linguistic Parse Tree from the Stanford Parser. (b) A Mapping Strategy for the Nodes in the Parse Tree. (c) A Valid Parse Tree. (d) A Query Tree after Inserting Implicit Nodes.

Query Tree Translator. Given the query tree verified by the user, the translator utilizes its structure to generate appropriate structure in the SQL expression and completes the foreign-key-primary-key (FK-PK) join paths. The result SQL statement may contain aggregate functions, multi-level subqueries, and various types of joins, among other things. Finally, our system evaluates the translated SQL statement against an RDBMS and returns the query results back to the user.

2.4 Parse Tree Node Interpretation

To understand the linguistic parse tree from the database's perspective, we first need to identify the parse tree nodes (words/phrases) that can be mapped to SQL components. Such nodes can be further divided into different types as shown in Figure 2.4, according to the type of SQL components they mapped to. The identification of select node, operator node, function node, quantifier node and logic node is independent of the database being queried. In NaLIR, enumerated sets of phrases are served as the real world "knowledge base" to identify these five types of nodes.

In contrast, name nodes and value nodes correspond to the meta-data and data, respectively, which entirely depend on the database being queried. Often, the words and phrases specified by the user are not exactly the same as the meta-data/data in the database. In Sec-

Node Type	Corresponding SQL Component
Select Node (SN)	SQL keyword: SELECT
Operator Node (ON)	an operator, e.g. =, <=, !=, contains
Function Node (FN)	an aggregation function, e.g., AVG
Name Node (NN)	a relation name or attribute name
Value Node (VN)	a value under an attribute
Quantifier Node (QN)	ALL, ANY, EACH
Logic Node (LN)	AND, OR, NOT

Figure 2.4: Different Types of Nodes in the Tokenization.

tion 2.4.2, we map these parse tree nodes to the meta-data/data in the database based on the similarity evaluation between them. Ambiguity exists when a parse tree node has multiple candidate mappings. In such a case, our system returns multiple candidate mappings for the user to choose from. In Section 2.4.3, we provide strategies to facilitate the user in recognizing the desired mappings from possibly many candidate mappings. Before that, in Section 2.4.1, we define the model we assume for the rest of the paper.

2.4.1 Data Model

In this chapter, a *relation schema* is defined as $R_i(A_1^i, A_2^i, ..., A_{k^i}^i)$, which consists of a *relation name* $n(R_i)$ and a set of *attribute schemas* $\{A_j^i | 1 \leq j \leq k^i\}$. A relation schema R_i has one (or a set of) attribute schema as its *primary key* $R_i.PK$ for identifying the tuples. A *database schema* D is a set of relation schemas $\{R_i | 1 \leq i \leq n\}$. Both relation schemas and attribute schemas are called *schema elements*. When populated with data, a database schema generates a database and each attribute A_j is populated with values $Vals(A_j)$. In natural language queries, naive users often informally refer to a tuple by specifying the value of one (or a set of) specific attribute. For example, naive users tend to refer to a paper by specifying its title. To capture this intuition, for a relation schema, we may use one (or a set of) attribute $R_i.PA$ as its *primary attribute* to identify tuples informally.

The *Schema Graph* $G(V, E)$ is a directed graph for a database schema D. V consists of two kinds of nodes: *relation node*s and *attribute node*s, corresponding to relation schemas and attribute schemas in D, respectively. Likewise, there are two types of edges in E: *projection edge*s and *foreign-key-primary-key (FK-PK) join edge*s. A projection edge starts from a relation schema R_i to each of its attribute schema $A_1^i, A_2^i, ..., A_{k^i}^i$, while a FK-PK join edge goes from a foreign key to a primary key when there is a FK-PK relationship between them. For each edge e in E, we assign a $weight$ $w(e)$, with a value between 0 and 1, where a larger

weight indicates a stronger connection. A simplified version of the schema graph for the Microsoft Academic Search database is shown in Figure 2.1, in which some nodes/edges are omitted.

A *join path* p is a list of schema elements, in which for every two adjacent schema elements v_i and v_j, (v_i, v_j) or (v_j, v_i) exists in E. The weight of p is defined as $w(p) = \Pi_{e_i \in p}(w(e_i))$. In this chapter, join paths containing the following pattern, $p \leftarrow f \rightarrow p$ where f is a foreign key and p is a primary key, are considered as invalid join paths.

2.4.2 Candidate Mappings

In a reasonable design of database schema, the names of schema elements should be meaningful and human-legible. Therefore, when the label of a parse tree node and the name of a schema element are similar in meaning or spelling, they are likely to correspond to the same real world object. To capture this intuition, we use the WUP similarity function [82], denoted as Sim_w, which is based on the Wordnet to evaluate the similarity between words/phrases in meaning. In addition, we adopt the square root of the Jaccard Coefficient between the q-gram sets of words/phrases, denoted as Sim_q, to evaluate their similarity in spelling [83]. Let $l(n)$ be the label of node n and $n(v)$ be the name of a schema element v. The name similarity function between $l(n)$ and $n(v)$ is defined as follows:

$$Sim_n(l(n), n(v)) = MAX \begin{cases} Sim_w(l(n), n(v)) \\ Sim_q(l(n), n(v)) \end{cases}$$

When their similarity is above a predefined threshold τ, we say that v is a candidate mapped schema element of n. Also, users may not be able to specify the exact values in the database. In our system, we use Sim_q to evaluate the similarity between the label of a parse tree node and a value. A value is a candidate mapped value of the parse tree node, if their similarity is above τ.

Definition 1 (NV Node) *A parse tree node, which has at least one candidate mapped schema element or candidate mapped value, is an NV node.*

Since a database often stores meta-data/data closely related to one another, many schema elements/values may be similar to one another, both in meaning and spelling. As a result, multiple candidate mappings may be returned, of which only a subset is correct. For example, in Figure 2.3 (a), the node "VLDB" may have multiple candidate mappings in the database of Microsoft Academic Search, say, VLDB, VLDB workshops, VLDB PhD work-

shop, PVLDB and VLDB Journal. In this case, it is very hard for the system to figure out which subset of the candidate mappings the user means.

We deal with this kind of ambiguity interactively with the user. For each ambiguous node, we return multiple of its candidate mappings for the user to choose from. To facilitate the user in recognizing the desired mappings from possibly many candidate mappings, we show candidate mappings hierarchically. In the first level, we show its top k_1 candidate mapped schema elements or the schema elements containing candidate mapped values. Then, under each of the k_1 schema elements, we show the top k_2 candidate mapped values. Users are free to choose a subset of the candidate mapping set as the final mappings. Note that all the final mappings must be of the same type, either in schema element or value. When all the final mappings are schema elements, the node is tokenized as a name node. Otherwise, it will be tokenized as a value node.

Given the vocabulary restriction of the system, some parse tree nodes may fail in mapping to any type of tokens. Also, some words cannot be directly expressed by SQL components. In such a case, a warning is generated, showing the user a list of nodes that do not directly contribute in interpreting the query. Our system deletes each such node from the parse tree and move all its children to its parent.

2.4.3 Default Mapping Strategy

To facilitate user choice, for each node, we would like to choose a mapping as the default mapping, which the user can simply accept in many cases.

A simple solution is to choose the mapping with the highest similarity. But sometimes, we can utilize the structure of the sentence, as reflected in the parse tree, to enhance the quality of the default mapping generation. Consider the query "return all conferences in the database area". The node "database" maps to both the value "database" under Domain.name and the value "database" under Keyword.keyword. Since the node "area" is the parent of the node "database" in the parse tree and maps to Domain with high similarity, the node "database" is more likely to refer to a domain name rather than a keyword. Our intuition is that when NV nodes are closer to each other in the parse tree, which means they are more relevant to each other, they should map to schema elements/values more relevant to each other in the database. The mutual relevance between schema elements is formally defined as follows:

Definition 2 (Relevance between two Schema Elements) *Given two schema elements v_1 and v_2 in the database, $p(v_1, v_2)$ be the join path connecting v_1 and v_2 with the highest weight. The weight of $p(v_1, v_2)$ is defined as the mutual relevance between v_1 and v_2.*

When we choose the default mappings for NV nodes, we consider both the similarity in the mappings and the mutual relevance between each pair of NV nodes. We define the score of a mapping strategy below. The mapping strategy with the highest score is returned as the default mapping strategy.

Definition 3 (Score of a Mapping Strategy) *Let M be a mapping strategy, in which each NV node n_i maps to the schema element v_{n_i}. Let $\{$ancestor$(n_i, n_j)\}$ be the set of all NV pairs where n_i is an ancestor of n_j and no NV node exists between n_i and n_j in the parse tree. The score of M is defined as follows:*

$$\Pi_{\{anc(n_i,n_j)\}}(Sim(n_i, v_{n_i}) * w(p(v_i, v_j)) * Sim(n_j, v_{n_j}))$$

2.5 Parse Tree Structure Adjustment

Given the correct mapping strategy, each node in the linguistic parse tree can be perfectly understood by our system. In this section, we infer the relationship between the nodes in the linguistic parse tree from the database's perspective and then understand the whole query. However, three obstacles lie in the way of reaching this goal.

First, the linguistic parse tree generated from an off-the-shelf parser may be incorrect. Natural language sentences describing complex query logics often have complex structures with modifier attachments, aggregations, comparisons, quantifiers, and conjunctions. As a result, the performance of an off-the-shelf parser is often unsatisfactory for such sentences. For example, the linguistic parse tree shown in Figure 2.3 (a) is a simplified output of the Stanford Dependency Parser, which incorrectly attaches "VLDB" to "Bob".

Second, the structure of the linguistic parse tree does not directly reflect the relationship between the nodes from the database's perspective. Consider the following three sentence fragments: (a) author who has more than 50 papers, (b) author who has more papers than Bob, and (c) author whose papers are more than Bob. The linguistic parse structures of these three sentence fragments are very different while their semantic meanings are similar from the database's perspective (describing the papers of the author are more than Bob/50). We need to make such relationships explicit and represent them properly.

Third, natural language sentences often contain elliptical expressions. As a result, even though we understand the relationship between all the explicit words or phrases, the sentence may still be ambiguous before the elliptical part is completed. Take the parse tree in Figure 2.3 (c) as an example. Although the relationship between each pair of nodes is clear, it still has multiple possible interpretations.

In this section, we describe the construction of the Parse Tree Structure Adjustor in detail, which is in charge of correcting the possible errors in the linguistic parse tree, making the relationships between existing nodes understandable from the database's perspective, inserting implicit nodes to the parse tree, and finally obtaining the exact interpretation for the whole query. When ambiguities exist, the parse tree structure adjustor will generate multiple candidate interpretations for the user to choose from.

2.5.1 Query Tree

Since our system is designed to be a query interface that translates natural language queries into SQL statements, the semantic coverage of our system is essentially constrained by the expressiveness of SQL. So, given a database, we represent our semantic coverage as a subset of parse trees, in which each such parse tree explicitly corresponds to a SQL statement and all such parse trees could cover all possible SQL statements (with some constraints). We call such parse trees as *Query Tree*s. As such, interpreting a natural language query (currently represented by a linguistic parse tree and the mapping for each its node) is indeed the process of mapping the query to its corresponding query tree in the semantic coverage.

We defined in Figure 2.5 the grammar of the parse trees that are syntactically valid in our system (all terminals are different types of nodes defined in Figure 2.4.). Query trees are the syntactically valid parse trees whose semantic meanings are reasonable, which will be discussed in Section 2.5.3, or approved by the user. Given the three obstacles in interpreting a linguistic parse tree, as we have discussed before, there is often a big gap between the linguistic parse tree and its corresponding query tree, which makes the mapping between them difficult. In our system, we take the following two strategies to make the mapping process accurate.

1	Q -> (SClause)(ComplexCondition)*
2	SClause -> SELECT + GNP
3	ComplexCondition -> ON + (leftSubtree*rightSubtree)
4	leftSubtree -> GNP
5	rightSubtree -> GNP|VN|MIN|MAX
6	GNP -> (FN + GNP) | NP
7	NP -> NN + (NN)*(Condition)*
8	condition -> VN | (ON + VN)

\+ represents a parent-child relationship

* represents a sibling relationship

Figure 2.5: Grammar of Valid Parse Trees.

First, our system explains a query tree in natural language, which enables the user to verify it. Query trees are intermediates between natural language sentences and SQL statements. Thus the translation from a query tree to a natural language sentence is quite straightforward, compared to that from a SQL statement [42].

Second, given a natural language query, our system will generate multiple candidate query trees for it, which can significantly enhance the probability that one of them is correct. The problem is that, when the query is complex, there may be many candidate query trees, which are similar to each other. To show the user more candidate query trees without burdening them too much in verifying them, we do the mapping in two rounds and communicate with the user after each round. In the first round, we return the top k parse trees, which are syntactically valid according to the grammar defined and can be obtained by only reformulating the nodes in the parse tree. Each such parse tree represents a rough interpretation for the query and we call them valid parse trees. In the second round, implicit nodes (if there is any) are inserted to the chosen (or default) valid parse tree to generate its exact interpretation. Our system inserts implicit nodes one by one under the supervision of the user. In such a way, suppose that there are k' possible implicit nodes in each of the k valid parse tree, the user only needs to verify k valid parse trees and k' query trees instead of all $k * 2^{k'}$ candidate query trees. Figure 2.3 (c) shows a valid parse tree generated in the first round, while this valid parse tree is full-fledged to the query tree in Figure 2.3 (d) after inserting implicit nodes.

2.5.2 Parse Tree Reformulation

In this section, given a linguistic parse tree, we reformulate it in multiple ways and generate its top k rough interpretations. To simplify the tree reformulation process, each logic node or quantifier node is merged with its parent. For example, in the parse tree of Query 3 in Figure 2.1, which is shown in Figure 2.7 (a), the quantifier node "each" is merged with its parent "area".

The basic idea in the algorithm is to use subtree move operations to edit the parse tree until it is syntactically valid according to the grammar we defined. The resulting algorithm is shown in Figure 2.6. Each time, we use the function $adjust(tree)$ to generate all the possible parse trees in one subtree move operation (line 6)[2]. Since the number of possible parse trees grows exponentially with the number of edits, the whole process would be slow. To accelerate the process, our algorithm evaluates each new generated parse tree and filter out bad parse trees directly (line 11 - 12). Also, we hash each parse tree into a number and

[2]There is an exception for the tree adjustment, in which a node "=" can be inserted to the Root node and the resulting parse tree will be directly added to the priority queue without evaluation.

store all the hashed numbers in a hash table (line 10). By checking the hash table (line 8), we can make sure that each parse tree will be processed at most once. We also set a parameter t as the maximum number of edits approved (line 8). Our system records all the valid parse trees appeared in the reformulation process (line 13 - 14) and returns the top k of them for the user to choose from (line 15 - 16). Since our algorithm stops after t edits and retains a parse tree only if it is no worse than its corresponding parse tree before the last edit (line 8), some valid parse trees may be omitted.

```
Algorithm 1: QueryTreeGen(parseTree)
1: results ← φ; HT ← φ
2: PriorityQueue.push(parseTree)
3: HT.add(h(tree))
4: while PriorityQueue ≠ φ do
5:    tree = PriorityQueue.pop()
6:    treeList = adjust(tree)
7:    for all tree' ∈ treeList do
8:       if tree' not exists in HT && tree'.edit < t then
9:          tree'.edit = tree.edit+1;
10:         HT.add(h(tree'));
11:         if evaluate(tree') >= evaluate(tree) then
12:            PriorityQueue.add(tree')
13:            if tree' is valid
14:               results.add(tree')
15: rank(results)
16: Return results
```

Figure 2.6: Parse Tree Reformulation Algorithm.

To filter out bad parse trees in the reformulating process and rank the result parse trees, we evaluate whether a parse tree is desirable from three aspects.

First, a good parse tree should be valid according to the grammar defined in Figure 2.5. We count the number of parse tree nodes that violate the grammar. The fewer invalid nodes in a parse tree, the better the parse tree is. For example, the parse tree in Figure 2.3 (a) has four invalid nodes, which are "paper", "more", "Bob" and "VLDB". Similarly, the parse tree in Figure 2.7 (a) has three invalid nodes: "citation", "most" and "total".

Second, the mappings between the parse tree nodes and the schema elements in the database can help to infer the desired structure of a parse tree. For example, in the parse tree shown in Figure 2.3 (a), the node "VLDB" is attached to the node "Bob", which is an incorrect modifier attachment. In the parse tree node mapper, we have mapped the node "VLDB" to Conference.name, "Bob" to Author.name and "paper" to Publication. As dis-

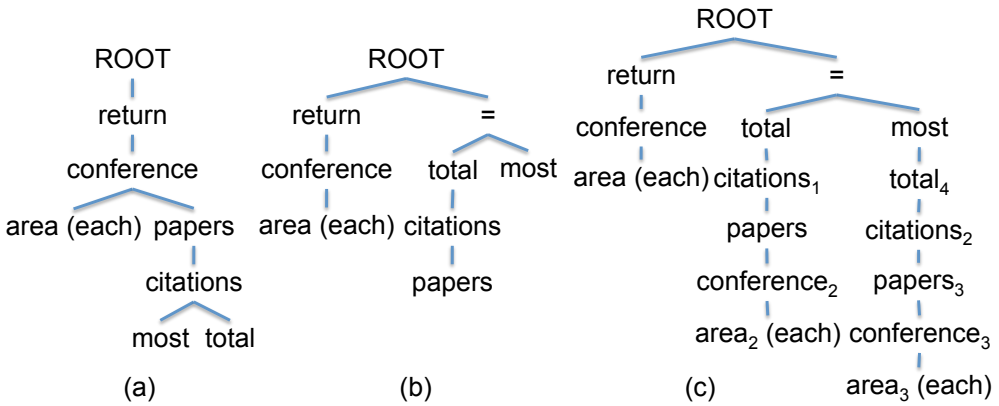


Figure 2.7: (a) A Simplified Linguistic Parse Tree for Query 3 in Figure 2.1. (b) A Valid Parse Tree. (c) A Query Tree after Inserting Implicit Nodes.

cussed in Section 2.4.3, in the database, Conference.name is more relevant to Publication than to Author.name. So the parse tree, which attaches "VLDB" to "paper", seems more reasonable than the parse tree attaches "VLDB" to "Bob". We capture such intuition by formally defining the score of a parse tree.

Definition 4 (Score of a Parse Tree) *Let T be a parse tree, in which each NV node nt_i maps to the schema element v_i. Let $\{valid(nt_i, nt_j)\}$ be the set of all the NV pairs where nt_i is an ancestor of nt_j and no NV node exists between nt_i and nt_j. Given the relevance $w(p(v_i, v_j))$ between v_i and v_i, the score of T is defined as follows:*

$$score(T) = \Pi_{\{valid(nt_i,nt_j)\}}(w(p(v_i, v_j)))$$

Third, the parse tree should be similar to the original linguistic parse tree, which is measured by the number of the subtree move operations used in the transformation.

When ranking the all the generated parse trees (line 15 in Figure 2.6), our system takes all the three factors into account. However, in the tree adjustment process (line 11), to reduce the cases when the adjustments stop in local optima, we only consider the first two factors, in which the first factor dominates the evaluation.

2.5.3 Implicit Nodes Insertion

Natural language sentences often contain elliptical expressions, which make some nodes in their parse trees implicit. In this section, for a rough interpretation, which is represented by a valid parse tree, we obtain its exact interpretation by detecting and inserting implicit nodes.

In our system, implicit nodes mainly exist in complex conditions, which correspond to the conditions involving aggregations, nestings, and non-FKPK join constraints. As can be

derived from Figure 2.5, the semantic meaning of a complex condition is its comparison operator node operating on its left and right subtrees. When implicit nodes exist, such syntactically valid conditions are very likely to be semantically "unreasonable". We will first use two examples to illustrate the concept of "unreasonable" conditions and then provide rules to detect and insert the implicit nodes, and finally make "unreasonable" conditions reasonable.

Consider two syntactically valid parse trees, whose semantic meanings are "return all the organizations, where the number of papers by the organization is more than the number of authors in IBM" and "return all the authors, where the number of papers by the author in VLDB is more than the number of papers in ICDE". The first parse tree compares the number of papers with the number of organizations, which sounds unreasonable in meaning. The second parse tree seems better, but compares the number of papers by an author in a conference with all the papers in another conference, which is also a little weird. To detect such unreasonable query logics, we define the concept of core node.

Definition 5 (Core Node) *Given a complex condition, its left (resp. right) core node is the name node that occurs in its left (right) subtree with no name node as ancestor.*

Inspired from [97], given a complex condition, we believe that its left core node and right core node are the concepts that are actually compared. So they should have the same type (map to the same schema element in the database). When they are in different types, we believe that the actual right core node, which is of the same type as the left core node, is implicit. Consider the first query we described above. The left core node is "paper" while the right core node is "author". By inserting an implicit node "paper" as its new right core node, the semantic meaning of the parse tree is changed to "return all the organizations, where the number of papers by the organization is more than the number of papers by the authors in IBM", which is much more reasonable. For the example in Figure 2.7, the implicit right core node "$citations_1$" in Figure 2.7 (c) is inserted in the same way. A special case is that when the right subtree contains only one number, there is no implicit node in the right subtree although the right core node is empty. For example, in the condition "more than 50 papers", the right subtree contains only one number "50" without any implicit node.

The name nodes in a left subtree are always related to the name nodes under the "SELECT" node. Take the parse tree in Figure 2.7 (b) as an example. The nodes "conference" and "area" are related to the nodes "citations" and "paper". In our system, we connect them by duplicating the name nodes under the "SELECT" node and inserting them to left subtree. Each of the nodes inserted in this step is considered as the same entity with its original node and marked "outside" for the translation in Section 2.6.

Furthermore, the constraints for the left core node and the right core node should be consistent. Consider the parse tree in Figure 2.3 (c). Its complex condition compares the

number of papers by an author in VLDB after 2000 with the number of all the papers by Bob (in any year on any conference or journal), which is unfair. As such, the constraints of "in VLDB" and "after 2000" should be added to the right subtree. To capture this intuition, we map each NV node under the left core node to at most one NV node under the right core node. Two nodes can be mapped only when they correspond to the same schema element in the database. When a node has no map, our system will add an implicit node to the other side to make them match with each other. Note that the nodes duplicated from "outside" nodes are also marked "outside" and are considered corresponding to the same entity with the original nodes.

The last kind of implicit node is the function node. In our system, we consider two cases where function nodes may be implicit. First, the function node "count" is often implicit in the natural language sentences. Consider the parse tree in Figure 2.3 (c). The node "paper" is the left child of node "more" and it maps to the relation "Publication", which is not a number attribute. The comparison between papers is unreasonable without a "count" function. Second, the function nodes operating on the left core node should also operate on the right core node. Figure 2.7 (c) shows an example for this case. We see that the function node "total" operates on the left core node "citations" but does not operate on the right core node "$citations_1$". Our system detects such implicit function node and insert "$total_4$" to the right core node.

In our system, the detection and insertion of implicit nodes is just an inference of the semantic meaning for a query, which cannot guarantee the accuracy. As such, the whole process is done under the supervision of the user.

2.6 SQL Generation

Given a full-fledged query tree, we show in this section how to generate its corresponding SQL expression.

In the translation, a schema element mapped by an NV node may have multiple representations in the SQL statement, say the schema element itself, its relation, its primary attribute and its primary key. For each occurrence, only one of these representations is adopted according to the context. Take the node "conference" in Figure 2.7 (c) as an example. When it is added to the FROM clause, it actually refers to the relation Conference. When we return it to the user as the final result, we actually return its primary attribute, which is Conference.name. For simplicity, we use the expression "the schema element mapped by node 'conference'" in all cases, in which the specific representation can be obtained from the context.

2.6.1 Basic Translation

In the cases when the query tree does not contain function nodes or quantifier nodes, which means the target SQL query will not have aggregate functions or subqueries, the translation is quite straightforward. The schema element mapped by the NV node under the SELECT node is added to the SELECT clause. Each value node (together with its operation node if specified) is translated to a selection condition and added to the WHERE clause. Finally, a FK-PK join path is generated, according to the schema graph, to connect each NV node and its neighbors. Such an FK-PK join path is translated into a series of FK-PK join conditions and all the schema elements in the FK-PK join path are added to the FROM clause.

Example 2 *Consider the query tree shown in Figure 2.7 (c). Here we omit its complex condition. The schema element Conference.name, which is mapped by the node "conference", is added to the SELECT clause. To connect the mapped schema elements Conference and Domain, a FK-PK join path from* $Conference$ *to* $Domain$ *through* $ConferenceDomain$ *is generated, which will be translated to the FK-PK conditions shown in line 16-17 in Figure 2.8.*

2.6.2 Blocks and Subqueries

When the query tree contains function nodes or quantifier nodes, the target SQL statements will contain subqueries. In our system, we use the concept of *block* to clarify the scope of each target subquery.

Definition 6 (Block) *A block is a subtree rooted at the select node, a name node that is marked "all" or "any", or a function node. The block rooted at the select node is the main block, which will be translated to the main query. Other blocks will be translated to subqueries. When the root of a block* b_1 *is the parent of the root of another block* b_2*, we say that* b_1 *is the direct outer block of* b_2 *and* b_2 *is a direct inner block of* b_1*. The main block is the direct outer block of all the blocks that do not have other outer blocks.*

Given a query tree comprising multiple blocks, we translate one block at a time, starting from the innermost block, so that any correlated variables and other context is already set when outer blocks are processed.

Example 3 *The query tree shown in Figure 2.7 (c) consists of four blocks:* b_1 *rooted at node "return",* b_2 *rooted at node "total",* b_3 *rooted at node "most", and* b_4 *rooted at node "*$total_4$*".* b_1 *is the main block, which is the direct outer block of* b_2 *and* b_3*.* b_3 *is the direct*

```
1. Block 2: SELECT SUM(Publication.citation_num) as sum_citation,
2.                  Conference.cid, Domain.did
3.          FROM Publication, Conference, Domain, ConferenceDomain
4.          WHERE Publication.cid = Conference.cid
5.                  AND Conference.cid = ConferenceDomain.cid
6.                  AND ConferenceDomain.did = Domain.did
7.          GROUP BY Conference.cid, Domain.did

8. Block 3: SELECT MAX(block4.sum_citation) as max_citation,
9.                  block4.cid, block4.did
10.         FROM (CONTENT OF BLOCK4) as block4
11.         GROUP BY block4.did

12.Block 1: SELECT Conference.name, Domain.name
13.         FROM Conference, Domain, ConferenceDomain
14.                 (CONTENT OF BLOCK2) as block2
15.                 (CONTENT OF BLOCK3) as block3
16.         WHERE Conference.cid = ConferenceDomain.cid
17.                 AND ConferenceDomain.did = Domain.did
18.                 AND block2.citation_num = block3.max_citation
19.                 AND Conference.cid = block2.cid
20.                 AND Conference.cid = block3.cid
21.                 AND Domain.did = block2.did
22.                 AND Domain.did = block3.did
```

Figure 2.8: Translated SQL Statement for the Query Tree in Figure 2.7 (c).

outer block of b_4. For this query tree, our system will first translate b_2 and b_4, then translate b_3 and finally translate the main block b_1.

For each single block, the major part of its translation is the same as the basic translation as we have described. In addition, some SQL fragments must be added to specify the relationship between these blocks.

First, for a block, each of its direct inner blocks is included in the FROM clause as a subquery. Second, each complex condition is translated as a non-FKPK join condition in the main query. Third, each of the name nodes that is marked "outside" refers to the same entity as its original node in the main block. Each of such relationships is translated to a condition in the main query.

Example 4 *Consider the query tree in Figure 2.7 (c), whose target SQL statement is shown in Figure 2.8 (the block b_4 is omitted since it is almost the same as b_2). In the query tree, b_4 is included by b_2 while b_2 and b_3 are included by b_1 as direct inner blocks. Thus their corresponding subqueries are added to the FROM clause of their direct outer blocks (line 10*

and line 14 - 15). The complex condition rooted at node "=" is translated to the condition in line 18. The nodes "$conference_2$", "$area_2$", "$conference_3$" and "$area_3$" are marked "outside" in the implicit node insertion, which means they correspond to the same entity as the nodes "conference" and "$area$" in the main block. These relationships are translated to the conditions in line 19 - 22.

Finally, we need to determine the scope of each aggregate function and quantifier (*all*, *any*). The scope of each quantifier is rather obvious, which is the whole block rooted at the name node marked with that quantifier. In contrast, when multiple aggregate functions exist in one query, especially when they occur in different levels of blocks, the scope of each aggregate function is not straightforward. In our system, we call the name node that are marked with "each" or "outside" as *grouping nodes*. Each aggregate function operates on the grouping nodes that (a) haven't been operated on by other aggregate functions in its inner blocks, and (b) do not have grouping nodes that meet condition (a) as ancestors. Once a grouping node is operated on by an aggregate function, it is disabled (since it has been aggregated). Each time we determine the scope of an aggregate function, we just add all the grouping nodes that still work. Take the query tree in Figure 2.7 (c) as an example. When determining the scope of $total_4$, both the grouping nodes $conference_3$ and $area_3$ work and are added to the GROUP BY clause (the same as line 7). After this aggregation, $conference_3$ is disabled since it has been operated on by $total_4$. When determining the scope of $most$, only $area_3$ still works, which will be added to the GROUP BY clause (line 11) and disabled afterwards.

2.7 Experiments

We implemented NaLIR as a stand-alone interface that can work on top of any RDBMS. In our implementation, we used MySQL as the RDBMS and the Stanford Natural Language Parser [24] as the dependency parser. For each ambiguity, we limited to 5 the number of interpretations for the user to choose from.

2.7.1 User Interface

Figure 2.9 shows a screenshot of NaLIR. When a new query is submitted from the client, the server processes it and returns the results. If NaLIR is uncertain in understanding some words/phrases, it adopts the best mapping as default and lists the others for the user to choose form. Also, when NaliR is uncertain in understanding the query intent behind the whole sentence, it lists multiple interpretations. To facilitate the user in recognizing her query

intent, NaLIR shows interpretations hierarchically, in which each cluster/interpretation is an approximate/accurate natural language description. Each time when a user makes a choice, NaLIR immediately updates its interpretations, evaluates the best interpretation and updates the results.

2.7.2 Measurement

The motivation of our system is to enable non-technical users to compose logically complex queries over relational databases and get perfect query results. So, there are two crucial aspects we must evaluate: the quality of the returned results (effectiveness) and whether our system is easy to use for non-technical users (usability).

Effectiveness. Evaluating the effectiveness of NaLIR is a challenging task. The objective in NaLIR is to allow users to represent SQL statements using natural language. Traditional IR metrics like recall and precision would not work very well since they will always be 100% if the translated SQL statement is correct and near 0% in many times when it is not. So, the effectiveness of our system was evaluated as the percentage of the queries that were perfectly answered by our system. (Note that this is a stiff metric, in that we get zero credit if the output SQL query is not perfect, even if the answer set has a high overlap with the desired answer). Since the situations where users accept imperfect/wrong answers would cause severe reliability problems, for the cases when the answers were wrong, we recorded whether the users were able to recognize such failures, whether from the answers themselves or from the explanations generated by our system. Also, for the failure queries, we analyzed the specific reasons that caused such failures.

Usability. For the correctly processed queries, we recorded the actual time taken by the participants. In addition, we evaluated our system subjectively by asking each participant to fill out a post-experiment questionnaire.

2.7.3 Experiments Design

The experiment was a user study, in which participants were asked to finish the query tasks we designed for them.

Data Set and Comparisons. We used the data set of Microsoft Academic Search (MAS). Its simplified schema graph and summary statistics are shown in Figure 2.1 and Figure 2.10, respectively. We choosed this data set because it comes with an interesting set of (supported) queries, as we will discuss next.

We compared our system with the faceted interface of the MAS website. The website has a carefully designed ranking system and interface. By clicking through the site, a user

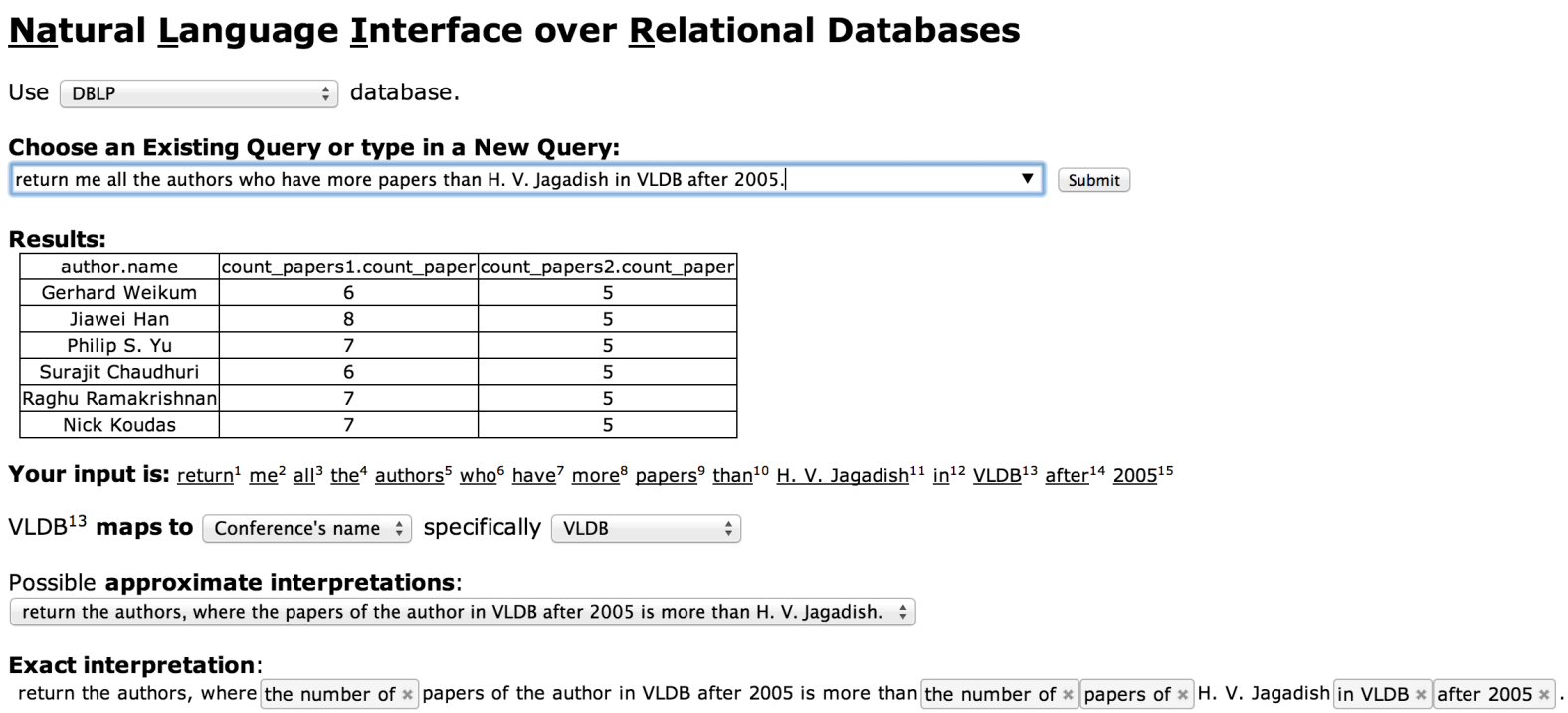


Figure 2.9: Query Interface of NaLIR

Relation	#tuples	Relation	#tuples
Publication	2.45 M	Author	1.25 M
cite	20.3 M	Domain	24
Conference	2.9 K	Journal	1.1 K
Organizations	11 K	Keywords	37 K

Figure 2.10: Statistics for MAS Database.

Easy: Return all the conferences in database area.
Medium: Return the number of papers in each database conference.
Hard: Return the author who has the most publications in database area.

Figure 2.11: Sample Queries in Different Complexity.

is able to get answers to many quite complex queries. We enumerated all query logics that are "directly supported" by the MAS website and can be accomplished by SQL statements. "Directly supported" means that the answer of the query can be obtained in a single webpage (or a series of continuous webpages) without further processing. For example, to answer the query "return the number of conferences in each area", the user has to look at 24 webpages, in which each webpage corresponds to the answer in an area. Thus this query is not considered to be directly supported by the MAS website. However, the query, Q_D, "return the number of conferences in the Database area" is a directly supported query. Queries that refer to the same relations and attributes but different values, are considered to have the same query logic. Thus, query Q_D has the same query logic as the query "return the number of conferences in the Graphics area". Through exhaustive enumeration, we obtained a set of 196 query logics.

We marked the complexity of each query according to the levels of aggregation/nesting in its corresponding SQL statement. Sample queries with different complexity are shown in Figure 2.11. In the query set, the number of easy/medium/hard queries are 63/68/65, respectively.

The MAS website is expressly designed to support these 196 query logics, and the user can click through the site to get to a results page, entering only values of constants into search boxes as needed along the way. We used this as the baseline for our comparison. In other words, how did natural language direction compare with click through, for the queries supported by the latter. (Note that an NLIDB supports many different queries beyond just these 196, while the website does not. We restricted our comparison to just the queries supported by the website).

A central innovation in NaLIR is the user interaction as part of query interpretation. To understand the benefit of such interaction, we also experimented with a version of NaLIR in

which the interactive communicator was disabled, and the system always chose the default (most likely) option.

Participants. 14 participants were recruited with flyers posted on a university campus. A questionnaire indicated that all participants were familiar with keyword search interfaces (e.g. Google) and faceted search interfaces (e.g. Amazon), but had little knowledge of formal query languages (e.g. SQL). Furthermore, they were fluent in both English and Chinese.

Procedures. We evenly divided the query set into 28 task groups, in which the easy/medium/hard tasks were evenly divided into each task group. This experiment was a within-subject design. Each participant randomly took three groups of tasks and completed three experimental blocks. In the first (resp. second) experimental block, each participant used our system without (with) the Interactive Communicator to accomplish the tasks in her first (second) task group. Then in the third experimental block, each participant used the MAS interface to do her third task group. For each task group, the participants started with sample query tasks, in order to get familiar with each interface.

For our system, it is hard to convey the query task to the participants since any English description would cause bias in the task. To overcome this, we described each query task in Chinese and asked users to compose English query sentences. Since English and Chinese are in entirely different language families, we believe this kind of design can minimize such bias. To alleviate participants' frustration and fatigue from repeated failures, a time limit of three minutes was set for each single query task.

2.7.4 Results and Analysis

Effectiveness. Figure 2.12 compares the effectiveness of our system (with or without the interactive communicator) with the MAS website. As we can see, when the interactive communicator was disabled, the effectiveness of our system decreased significantly when the query tasks became more complex. Out of the 32 failures, the participants only detected 7 of them. Actually, most of undetected wrong answers were aggregated results, which were impossible to verify without further explanation. In other undetected failures, the participants accepted wrong answers mainly because they were not familiar with what they were querying. In the 7 detected failures, although the participants were aware of the failure, they were not able to correctly reformulate the queries in the time constraint. (In 5 of the detected failures, the participants detected the failure only because the query results were empty sets). The situation got much better when the interactive communicator was involved. The users were able to handle 88 out of the 98 query tasks. For the 10 failed tasks, they only accepted 4 wrong answers, which was caused by the ambiguous (natural language) explanations gen-

	with Interaction	without Interaction	MAS
Simple:	34/34	26/32	20/33
Medium:	34/34	23/34	18/32
Hard:	20/30	15/32	18/33

Figure 2.12: Effectiveness.

	Mapper	Reformulation	Insertion	Translation
w/o Interaction	15	19	0	0
with Interaction	0	10	0	0

Figure 2.13: Failures in each Component.

erated from our system. In contrast, the participants were only able to accomplish 56 out of the 98 tasks using the MAS website, although all the correct answers could be found. In the failure cases, the participants were simply not able to find the right webpages, which often required several clicks from the initial search results.

Figure 2.13 shows the statistics of the specific components that cause the failures. We can see that our system could always correctly detect and insert the implicit parse tree nodes, even without interactive communications with the user. Also, when the query tree was correctly generated, our system translated it to the correct SQL statement. When the interactive communicator was enabled, the accuracy in the parse tree node mapper improved significantly, which means for each the ambiguous parse tree node, the parse tree node mapper could at least generate one correct mapping in the top 5 candidate mappings, and most importantly, the participants were able to recognize the correct mapping from others. The accuracy in parse tree structure reformulation was also improved when the participants were free to choose from the top 5 candidate valid parse trees. However, when the queries were complex, the number of possible valid parse trees was huge. As a result, the top 5 guessed interpretations could not always include the correct one.

Usability. The average time needed for the successfully accomplished query tasks is shown in Figure 2.14. When the interactive communicator was disabled, the only thing a participant could do was to read the query task description, understand the query task, translate the query task from Chinese to English and submit the query. So most of the query tasks were done in 50 seconds. When the interactive communicator was enabled, the participants were able to read the explanations, choose interpretations, reformulate the query according to the warnings, and decide to whether to accept the query results.

It is worth noting that, using our system (with interactive communicator), there was no

	with Interaction	without Interaction	MAS
Simple:	48	34	49
Medium:	70	42	67
Hard:	103	51	74

Figure 2.14: Average Time Cost (s).

instance where the participant became frustrated with the natural language interface and abandoned his/her query task within the time constraint. However, in 9 of the query tasks, participants decided to stop the experiment due to frustration with the MAS website. According to the questionnaire results, the users felt that MAS website was good for browsing data but not well-designed for conducting specific query tasks. They felt NaLIR can handle simple/medium query tasks very well but they encountered difficulties for some of the hard queries. In contrast, the MAS website was not sensitive to the complexity of query tasks. Generally, they welcomed the idea of an interactive natural language query interface, and found our system easy to use. The average level of satisfaction with our system was 5, 5 and 3.8 for easy, medium, and hard query tasks, respectively, on a scale of 1 to 5, where 5 denotes extremely easy to use.

2.8 Summary

We have described an interactive natural language query interface for relational databases. Given a natural language query, our system first translates it to a SQL statement and then evaluates it against an RDBMS. To achieve high reliability, our system explains to the user how her query is actually processed. When ambiguities exist, for each ambiguity, our system generates multiple likely interpretations for the user to choose from, which resolves ambiguities interactively with the user. The query mechanism described in this chapter has been implemented, and actual user experience gathered. Using our system, even naive users are able to accomplish logically complex query tasks, in which the target SQL statements include comparison predicates, conjunctions, quantifications, multi-level aggregations, nestings, and various types of joins, among other things.

CHAPTER 3

A Template-based Interface for Relational Databases

3.1 Introduction

Typically, an NLIDB has a semantic coverage, which consists of the supported query logics. To answer a natural language query correctly, first, the semantic coverage must be expressive enough to cover the desired query interpretation, and second, the system must be able to map the query to the correct point in the semantic coverage. The two requirements are not easy to satisfy at the same time since the difficulty of the mapping increases with the expressiveness of the semantic coverage.

First, the semantic coverage of a useful NLIDB is often required to be wide. Actually, in the scenario when the semantic coverage is narrow and predictable, form-based interfaces, faceted interfaces and WIMP (windows, icons, menus and pointers) interfaces are often preferred [61]. So, the real value of an NLIDB is its support for various types of complex queries over structurally complex databases. Based on this observation, some NLIDB systems try to define their semantic coverage as all the syntactically correct SQL queries (or XQueries) over the queried database [51, 47]. We notice that most syntactically correct queries are unnecessary in that they will almost never be queried by users. However, supporting them makes the mapping task much more difficult.

Second, natural language queries are inherently ambiguous. Even a grammatically correct natural language sentence may contain ambiguities, from the perspective of both natural language processing and database schema matching. Since the number of candidate interpretations grows exponentially with the number of ambiguities, the system's semantic coverage often contains multiple candidate interpretations. In such cases, it requires considerable intelligence to figure out which one is desired.

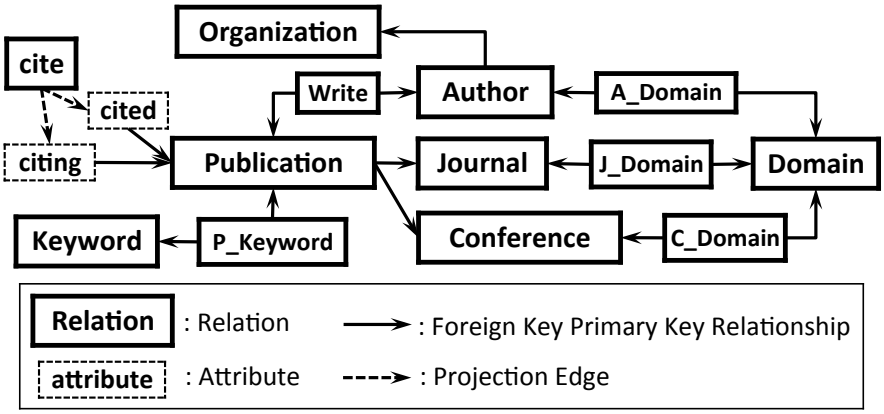


Figure 3.1: Simplified Schema Graph for Microsoft Academic Search

Example 5 *Consider the query "show me the citations of "Constructing an NLIDB..." that are not written by its authors" on the schema graph shown in Figure 3.1. Many ambiguities exist. First, the paper has a set of authors and each citation has a set of authors. It is not clear whether the query requires these two sets to be completely non-overlapping or different in at least one author. Second, two join paths exist when joining the two papers on the schema graph: (a)* $paper_1 \leftarrow citing \leftrightarrow cited \rightarrow paper_2$*, in which* $paper_2$ *is the reference, and (b)* $paper_1 \leftarrow cited \leftrightarrow citing \rightarrow paper_2$*, in which* $paper_2$ *is the citation. It is not easy to figure out which one is desired. Due to these ambiguities, the NL query can correspond to many SQL statements with semantics that are different, leading to different query results.*

Moreover, naive users often use informal expressions and prefer brevity to logical precision.

Example 6 *The query "authors who have 100 papers" usually means "authors who have greater than or equal to 100 papers". The greater than operator in the desired output query is nowhere in the input natural language expression: it has to be deduced from common sense and domain knowledge.*

Even though these ambiguities and shortenings are hard for a computer to resolve, we humans can usually tell which interpretation is correct since we can quickly disqualify most other interpretations as unlikely, according to our intuition and common sense. In this chapter, we would like to enable NLIDBs to have this kind of "intuition" by supporting query logics selectively. For example, for a bibliographic database, we would expect to support "which authors have more than 20 papers in SIGMOD?", but not necessarily to support "which papers have less than 2 authors in Stanford University?", even if these two queries are similar in syntax. The latter query is perfectly legal, and easily expressed in SQL. However, based on our domain knowledge and common sense, we can believe that it is an unlikely

query. As humans, when we hear a query like this, our brains go through an error correction process: Did I hear correctly? Could she have meant something else? Is there some other more sensible way I could interpret her question? We would like to have our system implement the corresponding skepticism. Similar ideas have previously been suggested, in the context of keyword queries [64].

Most natural language query systems generate multiple possible interpretations and use some kind of ranking or scoring mechanism to choose between them. Typically, this scoring is based only on the relevance of each candidate interpretation of the natural language query. In contrast, consider search engines, where a keyword query may have many relevant webpages. In most search systems, the ranking considers both relevance (e.g. using TF-IDF) and the importance of webpages (e.g. estimated using PageRank), as it achieves much higher precision and recall compared to ranking that is based solely on relevance [14]. Inspired by this observation, we model the semantic coverage of an NLIDB as a set of weighted *SQL templates*, in which the weight describes the likelihood of each template to be queried. Then, when ranking the candidate interpretations (templates) for a natural language query, we consider the weights of the interpretations as well as the relevance between the query and the interpretations.

The next immediate question is how to acquire these templates and learn the weights. In most systems, the query log records the queries issued by actual users. If a fairly large query log is available, then we can assume that the frequency with which any query logic appears in the query log reflects the probability of a future query being posed with the same logic. That is, the query log is a representative sample of the distribution of queries. Based on this idea, we can analyze the query log and define the concept of *popularity* for each SQL template as an estimate of its likelihood to be queried.

Ideally, when we have a large enough SQL query log, it is straightforward to define the popularity for each SQL template based on its frequency. However, in many real applications, the query log may not be enough to cover all necessary SQL templates. If the query log is only in natural language, then cost considerations may dictate that only a few queries are manually translated into SQL templates. Even if all SQL templates are covered, the query distribution, may not reflect the true distribution one would observe with an infinite (theoretical) query log. DBAs often have a good idea about the domain and typical user interest and are able to distinguish whether a template should or should not be supported based on their expertise. But it is not realistic to expect DBAs to enumerate all the SQL templates as each template requires time and effort to create.

To address these problems, we provide a strategy to augment the available resources such as query logs and specifications from DBAs. In our strategy, the existing SQL templates

are extended and a PageRank inspired mathematical model is developed for smoothing the popularity of each template.

Besides mapping to the desired SQL template, the ambiguities at entity level also need to be resolved. The identification for an entity used in natural language scenarios and that stored in the database are often different. In a database, keys are defined to uniquely identify entities, while in natural language scenarios, these identifications are often based on attribute values. However, attribute values stored in the database can be very similar, particularly when they are text strings, and users may not able to specify the exact value for the entity. We observe that in real world applications, important entities are more likely to be queried (e.g. authors with many papers, papers with many citations, and so forth). So we formally define the importance for the entities and take it into account for disambiguation at entity level.

Putting the above ideas together, we propose a framework for NLIDBs comprising two main parts: an offline part, which is responsible for generating weighted SQL templates by analyzing DBA input and the query log, and an online part, which is responsible for translating natural language queries into SQL queries, by mapping and instantiating the SQL templates. We have constructed such an NLIDB, and we call it TBNaLIR (Template-Based Natural Language Interface to Relational databases).

The intellectual contributions of this paper are as follows:

1. *Weighted SQL Templates.* We provide a generic method to model the semantic coverage of an NLIDB as a set of weighted SQL templates, in which each SQL template represents a set of SQL queries that differ only in values of constants and the weight of the template describes the likelihood of its being queried.

2. *System Architecture.* We provide a modular architecture for constructing NLIDBs, which consists of an offline part and an online part, in which each component can be designed, and improved, independently. We develop a working software system called TBNaLIR, which instantiates this architecture.

3. *Smoothing Model for Template Popularity.* We develop a simple probabilistic model, based on the behavior of a random user, to augment the available resources like query log and DBAs. Based on this model, we compute popularities for SQL templates, which better reflect the likelihood of each SQL template to be queried, even when the resources are limited.

4. *Mapping Strategy.* We present an effective mapping strategy to map a natural language query to the SQL templates. The mapping strategy considers not only the relevance

between the query and the templates, but also the popularity of the templates.

The remaining parts of the paper are organized as follows. In Section 4.2, we define the SQL template and overview our system architecture. We extend the semantic coverage of NLIDB in Section 3.3 and computes its popularity in Section 3.4. Section 3.5 discusses the online processing of a natural language query. In Section 4.6, our system is evaluated experimentally. In Section 4.7, we draw conclusions and point to future work.

3.2 Overview

The input to our system is a natural language query whose semantic meaning may involve comparisons, aggregations, various types of joins and nestings, among other things. The *semantic coverage* of our system is defined as a set of weighted SQL templates. In this chapter, each template is weighted by popularity, which describes the likelihood of a template being queried. Given a natural language query, by mapping it to the correct SQL template, our system translates the natural language query into the desired SQL statement and evaluates it against an RDBMS.

In this section, we first introduce SQL templates and then describe the system architecture.

3.2.1 SQL Template

Two queries are said to have the same *query logic* if they have the same SQL structure (SQL keywords, relations, attributes, aggregation functions, operators, nestings), even though constants in the query have different values. A *SQL template* is used to represent such a query logic by replacing the specific values in the SQL statement with slots. For example, the SQL template of "papers published after 2007 in VLDB" is shown in Figure 3.2.

```
SELECT *
FROM publication, conference
WHERE publication.cid = conference.cid
    AND conference.cid = [ID] AND publication.year > [INTEGER]
```

Figure 3.2: Sample SQL Template.

While the values of constants can be ignored in defining templates, it is important to distinguish between related operators. For example, the SQL templates describing "authors with $> x$ papers", "authors with $= x$ papers", and "authors with $< x$ papers" are considered as three different templates, since they have very different likelihoods of being queried. The

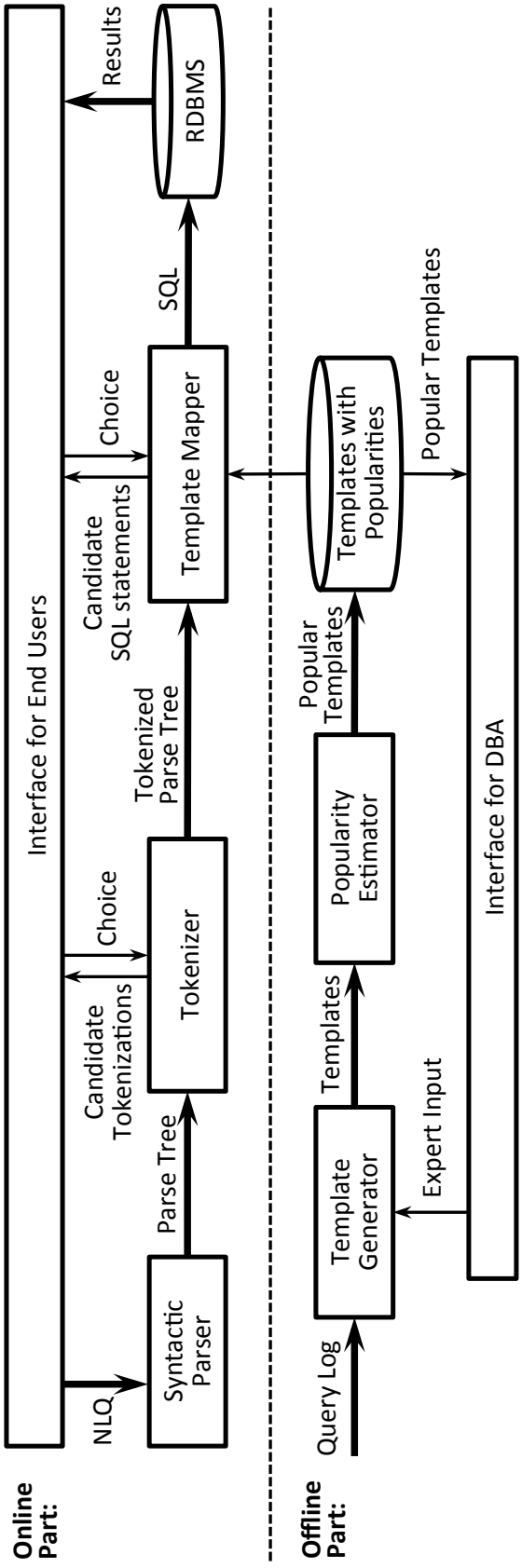


Figure 3.3: System Architecture for TBNaLIR.

first template is likely to be queried much more frequently than the other two. As such, it is unfair to define one big template to cover the three and mark all of them as high quality query logics. Similarly, SQL templates with different operators, quantifiers, aggregation functions, or logic keywords (AND, OR) should be considered as different SQL templates.

3.2.2 Problem Definition

The problem we attempt to address is to translate a natural language query (NLQ) to a SQL statement. In recent works, this problem is often modeled as a semantic parsing problem or machine translation problem, in which the translator is learned from a considerable number of (NLQ, SQL) pairs [26, 57, 54, 70]. However, unlike typical semantic parsing or machine translation problems, the underlying schema of different databases often differs, which means the training set for one NLIDB cannot be applied to another NLIDB. As a result, in real database applications, a good training set is often not available. That makes the configuration of an NLIDB hard, even if the learning strategy may work in theory. [99] focuses on the generation of training examples, but it still requires a set of natural language queries, which is often not available for relational databases.

Furthermore, unlike the general problem of semantic parsing or machine translation, an NLIDB has many unique resources for resolving ambiguities, like the underlying schema structure, the query log, the distribution of the data, the configuration from the DBA, and so forth. The fact that more resources can be taken into account often means higher dimensions of input, which exacerbates the shortage of training examples for a learning-based system. Instead, we observe that the essence of taking usage of these resources is that they reflect the likelihood of different query logic to be asked. Therefore, our solution is to capture this information about NLIDB structure and domain in the form of SQL templates.

Example 7 *Consider Example 15 where the two ambiguities mentioned would make up four different interpretations, which corresponds to four SQL templates. By checking the query log (if there is a fairly large one), it is very likely that the SQL template corresponding to the correct interpretation appears much more frequently than the other three. This information obtained from the query log could serve as a strong evidence for solving these ambiguities. Similarly, other resources such as the underlying data distribution can also serve as evidences for resolving some ambiguities. For example, consider the second ambiguity in Example 15 of generating join paths between two papers. When examining the data distribution of the underlying database, we could find that the number of references of a paper ranges from 0 to several hundred, while the number of citations of a paper ranges from 0 to tens of thousands. The distribution of the latter has a much larger entropy, which means a*

query on that is more informative. This evidence could also help us to resolve some of the ambiguities.

The *semantic coverage* of an NLIDB is the set of all questions that it can support. It may be possible to state natural language queries outside the semantic coverage, but the system would not be capable of answering these. We model the semantic coverage of an NLIDB as a set of weighted SQL templates, in which the weights represent the likelihood of each SQL template to be queried. Given this representation, the problem of understanding an NLQ is naturally modeled as a mapping problem, which tries to map the NLQ to its corresponding SQL template. The mapping is based on the both the relevance between the NLQ and the SQL template, and the weight of the SQL template.

The system is naturally divided into two parts. The offline part obtains the weighted SQL templates from available resources, including user logs and DBA directives. The goal is to make the SQL templates cover most of the likely NLQs and the weights to faithfully reflect their likelihood to be queried. The online part maps the input NLQ to the SQL templates. The goal is to improve the ranking to make the desired interpretation rank higher.

3.2.3 System Architecture

Figure 3.3 depicts the architecture of our system. It consists of two main components: an offline part and an online part.

3.2.3.1 Offline Part

The offline part, which consists of Template Generator and Popularity Estimator, is responsible for generating the weighted SQL templates as the semantic coverage from available resources.

Template Generator The template generator first transforms each SQL statement in the query log or specified by DBAs into a SQL template. This set of SQL templates forms the initial semantic coverage. Given the fact that these resources may be limited or biased, the initial semantic coverage may not be able to cover all (or most of) the likely queries. To deal with this problem, the template generator extends the initial semantic coverage by generating new SQL templates. This expanded set of SQL templates serves as a superset of the final semantic coverage of our system.

Popularity Estimator As will be discussed in detail in Section 3.3, the SQL templates are not independent of one another. Templates that share common structures are considered to

be relevant to each other. The Popularity Estimator uses this relevance relationship and a probabilistic model inspired from Pagerank to smooth the weight for each SQL template. The weight for each SQL template is defined as its popularity, which captures the following intuitions: (1) templates appear frequently in the query log tend to have high popularity, (2) templates relevant to many popular templates tend to be popular, and (3) If a SQL template is specified by an DBA that should (not) be supported, its weight and the weights of its relevant templates should increase (decrease). Finally, the SQL templates with high popularity are selected to form the semantic coverage of the NLIDB.

Interface for DBA The interface for DBA helps DBAs to manage the semantic coverage in a semi-automatic manner. The command of adding a new important SQL template may finally result in adding a series of SQL templates that are missing, while the command of deleting an existing popular template may decrease the weights of many its relevant SQL templates. It helps the DBAs do the modifications interactively, in which a few iterations would improve the semantic coverage a lot.

3.2.3.2 Online Part

The online part, which mainly consists of Tokenizer and Template Mapper, is responsible for mapping the natural language query to the correct SQL template and generating the SQL statement.

Syntactic Parser The first obstacle in translating a natural language query into a SQL query is to understand the natural language query linguistically. In our system, we use the Stanford Parser [24] to generate a linguistic parse tree from the natural language query. The linguistic parse trees in our system are dependency parse trees, in which each node is a word/phrase specified by the user while each edge is a linguistic dependency relationship between two words/phrases. We use dependency parser since it is regarded more stable for relation detection by capturing long-distance dependency [63].

Tokenizer In a dependency parse tree, each node is a single word. The tokenizer identifies the sets of nodes in the linguistic parse tree that can be mapped to database elements (SQL keyword, schema elements and values in the database). In this process, a set of words may be merged into a phrase, which should be taken as a whole to form a single token. Given the fact that important database elements are more likely to be queried, the mapping is based on the importance of elements, in addition to the similarity between phrases and the database elements.

Template Mapper The next step is to understand the relationship between the tokens and interpret the whole query. In the template mapper, a natural language query is interpreted by finding its corresponding SQL template. Specifically, our system ranks SQL templates based on both their popularity and their relevance to the input query. The top SQL templates are instantiated and explained to the user. The one chosen by the user (or the top one by default) is evaluated against the RDBMS.

Interface for End Users To deal with the possibility that the system possibly misunderstands the user, our system explains to the user how her query is processed. Specifically, interactive communications are organized in two steps, which verify the intermediate results of the tokenizer and the template mapper, respectively. For each ambiguity, our system generates a multiple choice selection panel, in which each choice corresponds to an interpretation. Each time a user changes a choice, our system immediately updates the results in later steps.

3.3 Template Generation

We notice that the syntactically valid SQL templates actually form an open set, in which most templates are not semantically meaningful and will almost never be queried by real users. Our goal is to develop enough SQL templates to cover all reasonable queries while ensuring that there are not too many unnecessary SQL templates. We develop these templates based on the query log and the specifications from the DBAs using a three-step process as follows:

1. **Initialization**: Obtain the initial SQL templates that directly appear in the query log or specified by DBAs.

2. **Extension**: Extending the initial semantic coverage by generating new SQL templates.

3. **Refinement**: Compute the popularity of the SQL templates and choose top ones as the final semantic coverage.

In this section, we discuss the first two steps. The refinement step will be described in the next section.

The initialization step is quite straightforward, according to the definition of SQL template. Given the fact that the queries in the query log may be limited or biased, not only can we be off on the frequencies of SQL templates that occur, it is also possible that we entirely miss templates that do occur in practice but happened not to be present in our small sample. This danger is exacerbated by our need to distinguish between operators and query structures

at fine granularity, which increases the number of SQL templates we need to support. Similarly, it is usually easy for an expert DBA to fast design some likely query templates, but it is very hard to be complete and make sure not to miss any likely query templates. So, in the extension step, the initial semantic coverage, whether obtained from query log or DBA, is extended by adding new SQL templates.

We define an *SPJA template* to be a SQL template that uses only selections, projections, FKPK (foreign-key-primary-key) joins and aggregations (including Group By and Having clauses). We first describe how SPJA templates are generated in the next several paragraphs, and then extend complex templates later in this section.

Example 8 *Given the simplified schema shown in Figure 3.1, the SQL template for the query in Example 15 is a complex (non-SPJA) template, which involves subquery with a NOT IN clause. The query in Example 6 can be expressed by an SPJA template.*

We start from the generation of SPJA templates. A similar problem arises in schema-based keyword search [37, 58, 31, 48]. However, we cannot simply adopt their method since too many SQL templates would be generated, in which most of them are semantically meaningless, especially when aggregations are considered.

Example 9 *Given a simple join path, paper - citing - cited - paper, many SQL templates can be generated. Here we choose two of them as examples, which correpsond to "papers with more than 1000 citations" and "papers with more than 1000 references", respectively. As humans with experiences in querying bibliographic databases, we could easily tell that the first one is much more likely to be queried than the second one. But how can we provide the system with such intuition?*

Here we take into account the underlying data distribution to distinguish semantically meaningful SQL templates from others. In Example 9, we observe that the number of citations for each paper ranges from zero to tens of thousands, while the number of references in each paper ranges from zero to several hundreds at most. Obviously, the former distribution is much more dispersed than the latter. In information theory, the entropy of the former would be higher, which means a query on that is more informative. We adopt the concept of entropy from information theory to estimate whether an SPJA template is semantically meaningful.

Definition 7 (Fanout Set) *Let R be all the schema elements returned, J be the FKPK join network at schema level, $\{r_i\}$ be the combination set of the tuples of R, j_r be the join networks corresponding to J at tuple level that contains the tuple combination r. The fanout set of the SQL template is defined as $M = \{|j_r| \mid r \in \{r_i\}\}$.*

Given the fanout set M, let $\{m\}$ be its distinct values and f_m be the relative frequency of m. The entropy of M is computed as follows:

$$E(M) = -\sum_{m \in M} f_m \cdot \log f_m \tag{3.1}$$

The entropy is then normalized to a number between 0 and 1, and a threshold is used to filter out the templates with low fanout entropies.

Given this concept, the system is automatically configured with a rough intuition of which SQL templates are more likely to be queried. For example, when generating the SQL templates containing two authors, the system is able to figure out that "the authors who coauthored with Bob" is more semantically meaningful than "the authors who are in the same organization as Bob", while "the authors who are in the same research domain with Bob" is essentially meaningless.

For complex templates, it is difficult to construct the special structures (non-FKPK joins, set operations, nestings) from scratch. Instead, we generate new complex templates only by modifying the complex templates that directly appear in the query log. Specifically, we modify the SPJ parts of complex templates by adding/deleting constraints. Consider the SQL template mentioned in Example 15 again. Given the SQL template of "show me the citations of 'Constructing an NLIDB...' that are not written by its authors", we are able to generate the SQL templates like "show me the citations of 'Constructing an NLIDB...' that are *published in VLDB* and are not written by its authors" by modifying its SPJ parts.

Additionally, we also allow DBA to specify templates that should NOT occur. This expert input can occasionally be valuable: for example, if the small query log has some idiosyncratic queries that are better ignored, or to guide our template set extension method described above by limiting some templates. These templates are simply recorded with a negative appearance at this stage. The weight will be smoothed in the next stage, described in the next section.

3.4 Popularity

Ideally we would like to have a sufficiently long query log that records the querying behavior of actual users. In this case, we can assume that the log includes all the necessary SQL templates with frequency proportional to the real distribution, and the appearance frequency of a SQL template can be used directly to reflect its chances to be queried again.

In practice, the resources available are often limited, and may be artificially derived as discussed above. In this case, we must worry about the query log not being a good represen-

tative sample of the entire distribution. In this section, we provide strategies to smooth the weights for the SQL templates, which makes them reflect their likelihood of being queried, even in the cases when the resources are limited.

3.4.1 Template Relevance

A central observation on which we base our smoothing model is that the SQL templates are not independent of one another. As such, if a template is considered important and has a high weight (e.g. appears frequently in the query log), other templates it ncludes, or those include it, should also be given some credit. We call such other templates *relevant* to the target template.

Example 10 *Consider the SQL templates show in Figure 3.4. Intuitively, if T_1 appears frequently in the query log, T_2 should also be assigned some weight. Similarly, if T_3 is specified by an DBA as one that should not be supported, the weight of T_4 should also be reduced.*

T_1: authors who have more than 10 papers in SIGMOD
T_2: authors who have more than 10 papers
T_3: paper with more than 5 authors
T_4: paper with more than 5 authors in Stanford
T_5: authors who have collaborated more than 10 papers with Bob

Figure 3.4: Pairs of Relevant Queries.

To capture this observation, two intuitions need to be quantified: (a) how to evaluate the relevance between two templates and (b) how to smooth the weights by transferring them from one template to its relevant templates.

In our system, two SQL templates are considered relevant only when the graph representation of one template contains/is contained by the other[1]. Let T_1 and T_2 be two SQL templates. In the cases when T_1 contains T_2, we define the relevance between them as $(\frac{1}{2})^{(\frac{|T_1|}{|T_2|})}$, in which $|T|$ is the size (total number of nodes and edges) of template T. This definition is based on the observation that the larger subgraph T_2 is of T_1, the more relevant they should be.

This is a rather rigid definition for relevance since only containing/contained relationship is taken into account. Indeed, the definition of the relevance between two templates is somewhat subjective, and the specific definition is not material for the rest of our framework. So

[1]Theoretically, the subgraph isomorphism problem is NP-complete [19], with running time that is exponential in the number of the graph nodes with the same label. In the graphs representing SQL templates, the number of such nodes is typically very small and the efficiency is not a problem.

a different choice of relevance function can easily be substituted. However, we point out that similarity through substitution is a bad idea, since that is precisely the type of difference we are trying to tease out, as indicated by the many examples we presented above. Inclusion, on the other hand, is a much stronger relationship.

As will be discussed in detail in the next section, we put all the SQL templates into a graph, in which two templates are connected by an edge if they are relevant to each other. A strategy inspired from PageRank is used to smooth the weights of SQL templates. Using this strategy, the weight of a SQL template can be transferred to another SQL template indirectly, even if they are not directly relevant.

Example 11 *Consider the SQL templates show in Figure 3.4.* T_1 *is relevant to* T_2 *since the graph representation of* T_1 *contains* T_2*. Similarly,* T_5 *is relevant to* T_2*. If* T_1 *appears frequently in the query log, some of its weight can be transferred to* T_5 *through* T_2*.*

3.4.2 Popularity of SQL Templates

In this subsection, we smooth the weight of templates based on following three intuitions:

1. A SQL template that appears frequently in the query log should be assigned a high weight.

2. A SQL template that shows high relevance with many important (high weight) SQL templates should also be assigned a high weight.

3. If a SQL template is specified by an DBA that should (not) be supported, its weight and the weights of its relevant templates should increase (decrease).

To capture the first two intuitions, we develop a random user query model inspired by PageRank, in which we pretend that after the user composes a SQL query, she is likely to randomly compose a series of follow up queries, each of which is related to the last query. Of course, we do not expect that real users will actually issue queries in this fashion. In fact, the order in which queries are issued is not of relevance for our purposes. Rather, what we want to do is to add queries to the limited query log that are relevant to the queries already in the log. This random follow up query model accomplishes this addition in a principled way.

Here, based on the relevance between SQL templates, we put all the SQL templates into one *Relevance Graph*.

Definition 8 (Relevance Graph) *The relevance graph* $G(V, E)$ *is an undirected graph, in which each node* T_i *in V is a SQL template. There is an edge* (T_i, T_j) *in* E*, if* T_i *and* T_j *are*

related. A weight $w(T_i, T_j)$ is set for each edge (T_i, T_j) to reflect the relevance between T_i and T_j.

In the random querying model, we assume there is a "random user" who composes her first SQL query by randomly choosing one SQL query in the query log. Then she keeps on composing a series of SQL queries, in which each SQL query is "related" to the last query she composed. Eventually, she gets bored and starts on another SQL query in the query log and composes another series of SQL queries. In this process, the probability that the random user submits the queries in a SQL template is defined as the popularity of that SQL template.

Suppose that after the random user composes a query in template T_i, the user has the probability of c to compose a follow up query and has the probability of $1 - c$ to get bored and start to choose a new SQL query in the query log. If she composes a follow up query, the likelihood of a SQL template T_j to be queried is proportional to the relevance between T_i and T_j. On the other hand, if she gets bored and choose another query in the query log, the likelihood of a SQL template T_j to be queried is proportional to its appearance frequency in the query log.

In formal notation, let $G = (V, E)$ be the relevance graph and L be the query log, in which a SQL template T appears $l(T)$ times. $l(T)$ is 0 if T never appears in the query log. Let T_i and T_j be two SQL templates and $w(T_i, T_j)$ be the relevance between them in G. $w(T_i, T_j)$ is 0 if T_i is not related to T_j. The probability that the user will query T_j after querying T_i is:

$$P(T_j|T_i) = c\frac{|w(T_i, T_j)|}{|\sum_{T_k \in V} w(T_i, T_k)|} + (1 - c)\frac{l(T_j)}{|L|}$$

Given the probability a SQL template T_j to be queried after the SQL template T_i, we define the template graph to compute the popularity for each SQL template.

Definition 9 (Template Graph) *The template graph $G_T(V, E)$ is a directed graph in which each node T_i in V is a SQL template. There is an edge (T_i, T_j) in E, if T_i and T_j are relevant or T_j appears in the query log. Specifically, each edge (T_i, T_j) is weighted by $P(T_j|T_i)$.*

Given the template graph, the computation of the popularity is quite straightforward, and very similar to the computation of PageRank for web pages. Let $Popularity$ be the vector of the popularities for each SQL template in the expanded semantic coverage. $Popularity$ is initialized according to the appearance frequency of each SQL template in the query log. Let $G_T(V, E)$ be the template graph, represented by an adjacency matrix. By multiplying $Popularity$ with $G_T(V, E)$ for a few rounds until its value converges, the popularity for each SQL template can be obtained. We then normalize each entry in $Popularity$ into a number

between 0 and 1 by first adding it to 1, then taking its binary logarithm, and finally dividing it by the maximum value in the vector.

When taking the specifications from the DBAs into account, $l(T_j)$ is modified accordingly. When the DBA specify that T_j should (not) be in the semantic coverage, we modify $l(T_j)$ by adding (subtracting) a big number. As such, when running the PageRank algorithm, a positive $l(T_j)$ could help T_j to absorb these weights from all the templates in the graph and distribute the weights to T_j's relevant templates. In a similar manner, a negative $l(T_j)$ could absorb weights from T_j's relevant templates and distribute these weights to all the templates in the graph.

3.5 Online Query Interpretation

Given an online natural language query represented by a dependency parse tree, we first interpret each of its words and phrases by mapping them to the database elements (SQL keywords, schema elements and database values). Then, the whole query is interpreted by finding its corresponding SQL templates from among those generated in the offline part.

3.5.1 Element Mapping and Tokenization

We first identify the parse tree nodes (words/phrases) that can be mapped to database elements. Such nodes can be further divided into different types as shown in Figure 3.5, according to the type of database elements they mapped to. The identification of select node, operator node, function node, quantifier node and logic node is independent of the database being queried. Following [51, 47], we enumerate sets of phrases as the database independent lexicon to identify these five types of nodes.

Node Type	Corresponding SQL Component
Select Node (SN)	SQL keyword: SELECT
Operator Node (ON)	an operator, e.g. =, <=, !=, contains
Function Node (FN)	an aggregation function, e.g., AVG
Name Node (NN)	a relation name or attribute name
Value Node (VN)	a value under an attribute
Quantifier Node (QN)	ALL, ANY, EACH
Logic Node (LN)	AND, OR, NOT

Figure 3.5: Types of Nodes.

In contrast, name nodes and value nodes correspond to the schema elements and tuples in the underlying database, respectively. Often, the words/phrases specified by the user are not exactly the same as the ones stored in the database. In previous literature, the mapping is typically based only on the similarity between the words/phrases, with possible thesaurus expansion, and the schema elements/tuples in the database. However, one should expect that important schema elements/tuples are more likely to be queried. For example, an author name specified by the user may have multiple mappings in a bibliography database. The important ones (with many papers with many citations) are more likely to be the one in the user's mind. In this chapter, we formally capture this intuition and define the importance for the schema elements and tuples in the database.

Definition 10 (Tuple Importance) *Represent the given database as a directed data graph, in which each node is a tuple in the database and each edge is a foreign-key-primary-key (FKPK) relationship between two tuples*[2]*. The unnormalized importance of each tuple is defined as* $log_2(pagerank)$*, in which pagerank is its pagerank [65].*

Definition 11 (Schema Element Importance) *Let* S *be the semantic coverage generated in the offline part of our system. The unnormalized importance of each schema element is defined as* $(\sum_T Popularity(T))$*, in which* T *iterates over all the SQL templates in* S *that contain the schema element.*

The importance of each tuple (resp. schema element) is then normalized to a number between 0 and 1 by dividing it by the maximum importance of any tuple (resp. schema element) in the database.

In this chapter, we map the nodes to the schema elements and tuples based on both similarity and importance. Let n be a word or a phrase and v be a schema element or a tuple. The goodness of the mapping between them is scored as follows:

$$Score(n, v) = Similarity(n, v) + Importance(v)$$

Specifically, the similarity between a node and a schema element is defined as the max of their semantic similarity (WUP similarity [82] based on wordnet) and their spelling similarity (square root of the coefficient of their q-gram sets [83]). The similarity between a node and a tuple considers only their spelling similarity, for efficiency reasons.

For each node, the best mapping is set as the default interpretation for the node, but the user is shown the top k mappings as alternatives to choose from. Given the vocabulary

[2]To avoid the situation that the importance gets "trapped" in tuples with no outgoing links, we add both a PK-FK edge and an FK-PK edge in the data graph to represent an FKPK relationship in the database.

restriction of the system, some parse tree nodes may fail in mapping to any type of tokens. In such a case, a warning is generated, showing the user a list of nodes that do not directly contribute in interpreting the query. Our system deletes each such node from the parse tree and moves all its children to its parent.

3.5.2 Template Mapping

In this section, we interpret the whole query by mapping it to the SQL templates generated in the offline part. Given a natural language query NLQ, represented by a tokenized parse tree, its mapping score to a SQL template T is define as:

$$Score(NLQ, T) = Relevance(NLQ, T) + Popularity(T)$$

Specifically, $Popularity(T)$ is the popularity defined in Section 3.4.2. $Relevance(NLQ, T)$ is the relevance between NLQ and T. Given the fact that NLQ is represented by a tokenized parse tree, in which trivial nodes have been deleted after the tokenization, most of the information specified in NLQ should be contained by T, if it is relevant to NLQ. Reciprocally, one would think most of the information in T should also be contained in NLQ. However, natural language queries tend to be brief and often leave out things that "can be understood". Furthermore, natural language queries typically do not include schema-specific constructs such as FK-PK join paths, which are often by-products of normalization. So, in the reciprocal direction, we expect that most of the *major* information in T should also be contained in NLQ, according to a definition of *major* that we provide in the next paragraph. Putting these ideas together, NLQ is considered relevant to T if (a) the information in NLQ is contained by T, and (b) the major information in T is specified in NLQ. In particular, we define their relevance as follows:

$$\frac{|Info(NLQ) \cap Info(T)|}{|Info(NLQ)|} * \frac{|MInfo(NLQ) \cap MInfo(T)|}{|MInfo(T)|}$$

We define $Info(NLQ)$ as the set of ($parent$, $child$) relationship in NLQ. $Info(T)$ as the set of ($node_i$, $node_j$) in T, in which $node_i$, $node_j$ are ***referred nodes*** in NLQ that are *directed related*. In this chapter, referred nodes are the nodes in T that are referred to by the query NLQ. Directly related means there is a path between $node_i$ and $node_j$, which is not interrupted by other referred nodes. $MInfo(NLQ)$ is the set of nodes in NLQ, while $MInfo(T)$ is all the ***major nodes*** in the SQL template T. The major nodes are the nodes whose information is important and unlikely to be omitted by the user. For example, MAX is a major node, which is not likely to be omitted by the user, while $COUNT$ is not a major

Relation	#tuples	Relation	#tuples
Publication	2.45 M	Author	1.25 M
cite	20.3 M	writes	6.02 M
Conference	2.9 K	Journal	1.1 K
Organizations	11 K	Keywords	37 K

Figure 3.6: Statistics for MAS.

node, since it is often implicit in the user's query. We enumerate the set of major nodes, which serves as the knowledge base that can be used independent of domains.

3.6 Experiments

In our system, the quality of the semantic coverage, which is automatically generated in the offline part, directly affects the behavior of the online part. As such, we first evaluate the offline part separately and then show the benefits it brings for the online part.

Dataset. The dataset we use is Microsoft Academic Search (MAS), whose simplified schema graph is shown in Figure 3.1. Some summary statistics of its major relations are shown in Figure 3.6.

3.6.1 Offline Part

The goal of the offline part is to generate a high quality semantic coverage, in which most of the necessary query logics are covered, while including as few additional (unnecessary) query logics as possible. Adopting standard terminology from information retrieval, we would like to create a set of query templates that has very high recall (coverage is good) and acceptable precision (not too many unnecessary templates in the set). Since, practically speaking, our direct concern is with the size of the set created, we report this size directly, rather than the precision that it is roughly proportional to.

Test Set (Gold Standard). To evaluate the recall of a semantic coverage, we need a gold standard. The old version interface of MAS dataset was a carefully designed faceted interface. By clicking through the site, a user is able to get answers to many quite complex queries. We enumerated all query logics that are directly supported by the MAS website and obtained a set of 196 SQL templates. We noticed that the semantic coverage of the MAS website was constrained by the overall design of the interface. As a result, some likely query logics are not supported because they would require a redesign of the interface and impact

the overall visual effect. Also, it is possible that some unnecessary query logics are supported just because they are more compatible with interface. This is a common limitation of forms-based interfaces, faceted interfaces and WIMP (windows, icons, menus and pointers) interfaces. To address these shortcomings, we asked five domain experts, who often do academic searches and have a good sense of which questions would be frequently asked over a bibliographic database, to modify this set of query logics, deleting unnecessary ones and adding new ones as needed. In this process, we accepted a modification only when it had the agreement of all five experts. The final test set Q_{gold} we obtained is composed of 160 SQL templates, of which 105 were supported by the MAS website. We use this as the gold standard to evaluate the auto-generated semantic coverage.

Query Log. Unfortunately, MAS does not make its query log public. Therefore, we created our own (small) query log from 20 PhD students majoring in computer science or information science. All of them have the need to do academic searches and are interested in answers to complex queries over the MAS dataset. Each of them was asked to specify 15 SQL queries (we offered technical help with SQL as needed, but were careful not to suggest query interpretations or semantics). Thus we obtained a query log comprising 300 queries. This query log covers 109 out of the 160 SQL templates in Q_{gold}.

Semantic Coverage Extension. We extend this query log by generating additional SQL templates. The SQL templates generated in this process actually form an open set. So we stop when the number of SQL templates reaches 1000. In this process, when preferring the extensions with high normalized entropies, the top 1000 SQL templates covers 154 out of the 160 SQL templates in Q_{gold}. We compare with the baseline strategy used in [92, 31], which measures the queribility of a query logic by the length of the join path. When preferring the extensions with short join paths, the top 1000 SQL templates only include 133 SQL templates in Q_{gold}.

Popularity Estimation. As discussed, except when the query log is fairly large, the frequency of each SQL template appears in the query log may not be a good representative for the entire distribution, especially in the cases when many of the necessary SQL templates are automatically generated. So we smooth the popularity for the SQL templates in the semantic coverage. After the popularity is smoothed, 152 SQL templates out of the 154 SQL templates in the semantic coverage are ranked in the top 300, which means the necessary SQL templates generally gain high popularities in the smoothing. Remember that our system could also augment the information provided from the DBAs. In the cases when a DBA is allowed to modify the automatically generated semantic coverage, the top 300 SQL templates could cover the entire gold standard test set after only two key SQL templates are added. The detailed results are shown in Figure 3.7, where the x axis denotes the size of

(total number of SQL templates in) the semantic coverage and y axis is the number of the SQL templates in Q_{gold} that are covered.

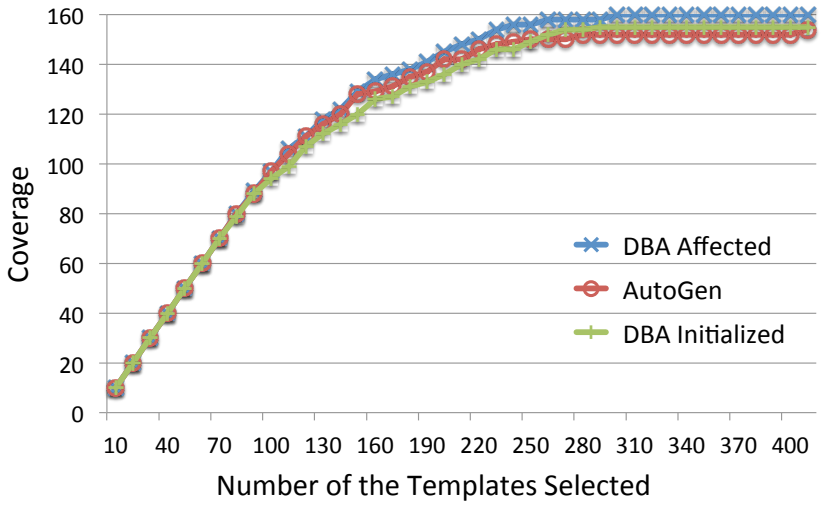


Figure 3.7: Quality of the Semantic Coverage.

Initial Templates provided by Domain Experts. When there is no query log available, a domain expert (such as the DBA) could specify the initial SQL templates. We note that it is difficult for a domain expert to be complete in their specification of all required templates, while it is not difficult for the expert to specify several popular templates. Therefore, extension and smoothing is expected to make a big difference. To evaluate this, we asked a domain expert, who often does academic searches and has a good intuition of which questions are often asked in a bibliographic database, but was not involved in the creation of Q_{gold}, to examine the database schema and specify the initial SQL templates. We got an initial set comprising 50 distinct SQL templates, of which 48 of were in Q_{gold}. After the SQL template extension, 155 SQL templates in Q_{gold} were covered in the top 1000 SQL templates. After smoothing the weights, all 155 SQL templates were ranked in the top 300 SQL templates. Detailed results are shown in Figure 3.7.

3.6.2 Online Part

We implemented the online part of our system as a stand-alone interface that can work on top of any RDBMS. In our current implementation, we used MySQL as the RDBMS and the Stanford Parser [24] as the dependency parser.

The online part of our system translates natural language queries into SQL statements. The aspect we must evaluate is the precision in the translation. We notice that this precision depends on the specific expressions used in describing the queries. Even for describing

the same query, the natural language expressions used by naive users are often different from those specified by technical users: naive users tend to use informal expressions and prefer brevity to grammatical correctness and logical precision. So, we designed user study experiments for the online part, in which non-technical participants are recruited to specify queries in their preferred natural wording.

Participants. Sixteen Participants were recruited with flyers posted on a university campus. A questionnaire indicated that all participants were familiar with keyword search interfaces (e.g. Google), but had little knowledge of formal query languages (e.g. SQL). Furthermore, they were fluent in both English and Chinese. For our experiments, it is hard to convey the query task to the participants since any English description would cause bias in the task. To overcome this, we described each query task in Chinese and asked users to compose English query sentences. Since English and Chinese are in entirely different language families and we believe this kind of design can minimize such bias.

Query Task Design and Procedures. As mentioned in Section 3.6.1, the curated MAS query set Q_{gold} we used in the offline part consists of the query logics that regarded by domain experts as necessary. Moreover, Q_{gold} contains various kinds of complex queries, in which 110, 52, 53, 34 queries contain aggregations, long join paths with length ≥ 5, subqueries, and multilevel of subqueries, respectively. As such, we use Q_{gold} to test the online part of our system. We transform each of the query logics into a query task and evenly divide the query tasks into 16 task groups, in which each task group contains 10 query tasks. Each participant randomly takes one task group and completes it though our system. For each task group, the participants start with sample query tasks, in order to get familiar with each interface.

Our system is designed to be robust. Even in the cases when the users input is just some keywords, for example, "author paper most", our system is still very likely to interpret it as "author with the most papers" rather than "paper with the most authors", since the SQL template of the former has a higher popularity. As such, at the beginning of each query task, users are encouraged to specify their queries in the way they like. The initial expression for each query task is recorded for comparative experiment, since any reformulation would be affected by the feedback of our system.

Measurement. We evaluate effectiveness of the top 1 interpretation returned as $\frac{|M_P|}{|M_T|}$, in which $|M_T|$ is the total number of queries and $|M_P|$ is the number of queries that can be directly translated to the correct SQL statements. Remember that, when ambiguity exists, our system also returns alternative interpretations for the user to choose from. As such, we also evaluate the effectiveness of the top 5 interpretations returned as $\frac{|M_R|}{|M_T|}$, in which M_R is the number of queries in which one of the candidate interpretations returned is correct. Specifically, our system returns the user at most five candidate interpretations.

	Top 1 Result	Top 5 Results
(a) Initial	69	93
(b) Extended	74	104
(c) Smoothed	93	127
(d) DBA Affected	101	135
(e) Initialized by DBA	92	130
NaLIR	84	101

Figure 3.8: Experimental Results for the Online Mapping.

Comparisons. Our system models the natural language interpretation problem as a mapping problem, which maps the natual language query to the SQL templates in our semantic coverage. As such, for a query task, if its corresponding SQL statement is not in the semantic coverage, the online part of our system can never handle it correctly. As mentioned, the semantic coverage of our system is first extended and then refined based on the popularity computed generated. Also, it can be modified by the DBAs in a semi-automatic way. So in the comparative experiment, we test our system on five semantic coverages, which are (a) the initial semantic coverage that consists of the SQL templates directly appear in the query log, (b) the extended semantic coverage, in which additional SQL templates are generated but the weights are not smoothed, (c) the refined semantic coverage, in which the weights of SQL templates are smoothed and popular ones are selected as the semantic coverage, (d) the semantic coverage that are affected by the DBA after only two edits, and (e) the semantic coverage that is generated from the initial SQL templates provided by the DBA. We also compare our system with NaLIR [47], a generic NLIDB whose semantic coverage is the syntactically valid SQL statements (under some syntactical constraints).

The experimental results are shown in Figure 3.8. In experiment (a), we use query log L mentioned in Section 3.6.1 as the semantic coverage directly. The weight of each SQL template is set as its appearing frequency with simple normalization by first adding one and then taking the log. This semantic coverage only covers 109 out of the 160 queries in the test set, which heavily limits the performance of the online part. In experiment (b), the semantic coverage is extended from L, which contains 1000 SQL templates and covers 154 queries of the testing set, in which new added SQL templates are all considered as appearing once. The performance improves but is still limited, since the unnecessary SQL templates disturb the mapping. In experiment (c), the mapping improves a lot after the semantic coverage is refined, which contains 300 SQL templates and covers 152 queries of the testing set. In this process, the popularity of each SQL template is computed and the SQL templates

with low popularity are filtered out. In experiment (d), as mentioned in Section 3.6.1, after adding two key SQL templates, the semantic coverage covers all the test queries and the performance further improves. In experiment (e), the semantic coverage is generated from the SQL templates provided by the DBA, which behaves similar to that generated from a query log.

3.7 Summary

In this chapter, we provide a generic method to model the semantic coverage of an NLIDB as a set of weighted SQL templates, in which each SQL template represents a set of SQL queries in the same logic while the weight describes the likelihood of each SQL template to be queried. Given a natural language query, by mapping it to the correct SQL template in the semantic coverage, the query can be translated into the desired SQL statement, which may include comparison predicates, conjunctions, quantifications, multi-level aggregations, nestings, and various types of joins, among other things. We develop a simple probabilistic model for the behavior of a random user, and use this to augment the available query log. Based on this model, we compute the popularity for SQL templates, which better reflects the likelihood of each SQL template to be queried, even when the queries in the query log are limited or biased. We provide a modular architecture for constructing NLIDBs, which consists of an offline part and an online part, in which each component can be designed, and improved, independently. The framework described in this chapter has been implemented, and actual user experience gathered. Using our system, a small sized query log is enough to generate the necessary SQL templates, and even naive users are able to accomplish logically complex query tasks against our system.

CHAPTER 4

Interpreting Natural Language Queries over Databases

4.1 Introduction

A central challenge in building an NLIDB is the huge variability in natural language expressions: there are many ways to say the same thing, with differences possibly not only in the specific words used but even in the structure of the sentences. In consequence, it is unlikely that a simple rule-based translator can handle this huge variation. This sort of variability in natural language has been a central challenge in semantic parsing and machine translation for many years. Recently, deep learning has generated much enthusiasm as being able to *learn* these hidden structures. In recent works along these lines, understanding a natural language query is often modeled as a semantic parsing problem or machine translation problem, in which the translator is learned from (NLQ, SQL) pairs of training examples [26, 57, 54, 70]. In this strategy, the potential output forms an open set, which covers all the possible semantic meaning representations that are syntactically valid. In addition, the expressions describing the same query logic can be various. As a result, a considerable number of training examples are needed. While obtaining such training data may be challenging, it can at least be considered, for a specific database schema of interest. But unlike typical semantic parsing or machine translation problems, the underlying schema of different databases often differs, which means the training set for one NLIDB cannot be applied to another NLIDB. As a result, in real database applications, enough training data is virtually impossible to collect. Furthermore, any learned translator will work only until the next schema update, after which additional training is required.

Example 12 *Given a bibliographic database with schema shown in Figure 4.1, "authors with h-index over 30" and "authors who have more than 30 papers, in which each of these*

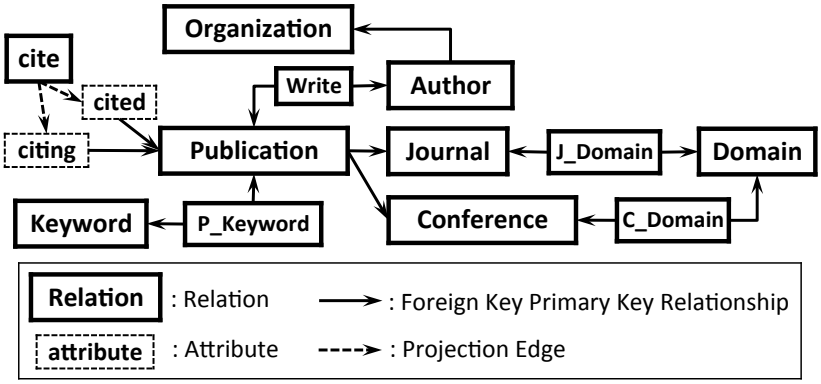


Figure 4.1: Schema Graph for MAS

papers have more than 30 citations" are two expressions for the SQL template in Figure 4.3.

Example 13 *When users using natural language interfaces, some of them would prefer brevity and specify queries like "who have more papers in SIGMOD than Feifei Li". Others might be aware that they are interacting with computers and try to specify a more completed query on purpose like "show me the authors, in which the number of papers on SIGMOD by the author is more the number of papers on SIGMOD by Feifei Li". Both of these expressions might be hard for most NLIDB to process since the first one has many ambiguities while the second one is too long for most parser to parse correctly.*

We observe that, given a database, the potential queries are often predictable, since the questions that can be answered are constrained by the schema and the data of the database. As such, the system does not need to support any query beyond that scope. In fact, even from out of this set of questions that could be asked, the ones that are actually asked constitute only a small fraction. This observation has been used to better interpret keyword queries in [64], and has even been considered in the context of NLIDB in [71]. In other words, we can model the semantic coverage of an NLIDB as a set of SQL templates. Thus, the NL query translation problem becomes one of matching a query to its corresponding SQL template. If we follow traditional NLIDB design, this matching is performed through a set of mapping rules, with the corresponding difficulties in dealing with structural variance. Our idea, in this paper, is to avoid having the system understand every detail of the natural language query and compose the SQL statement from the ground. Rather, we perform a coarse matching to find the relevant SQL templates. Where multiple matching SQL templates are found, we resort to user feedback to help choose the correct one.

While coarse matching can handle some amount of variability in natural language query expression, by itself it may not be sufficient to address radically re-structured possible natural

language query expressions. One way to address this problem is exploit a log of previously issued natural language queries. If we can find a similar query that was issued previously, then we can follow in its footsteps to find the correct matching SQL template. This method actually turns out very well in practice, because the many variants of natural language queries usually cluster nicely, with queries within a cluster differing primarily in choice of synonyms and in the occurrence of extra stop words.

The immediate next question is how to capture the similarity between natural language expressions. Early works like TF-IDF dealt with word occurrence frequencies in a principled way but did not capture the similarity between individual synonyms. Recent works attempt to learn a latent low-dimensional representation of sentences [45] from a generic corpus. This strategy may work well in document matching. However, in our situation, all the natural language queries are focused on a narrow domain (the underlying database). Sentences with different semantic meanings might be embedded very close to each other. Other metrics with hyperparameters, which try to capture the complex structure of natural language, may be hard to configure, given the difficulty of gathering enough training data on the querying domain.

As such, the metrics we present are hyperparameter free, which means they can be used even in the cases when the training examples are very limited. Specifically, our approach leverages the recent results of word2vec [60], in which vector representation for each word using a neural network language model. After the word embedding, semantically similar words are close to each other in the space. Leveraging this idea, we represent each word as a low-dimensional vector and model each sentence as a collection of low dimensional vectors. The distance between two sentences can then be computed as the minimum cost of transforming this representation of one sentence to a corresponding representation of the other sentence.

In determining the cost of such transformation, the influence of common words (and possibly even stop words) could be the same as that of distinctive meaningful words. To deal with this problem, we borrow ideas from TF-IDF in the distance computation of word vectors. Thereby, expressions that differ only in stop words, word order or synonyms can be found to be close to each other.

All of the above can work if we have a good log of past user queries. However, no such log may be available when the system is initially deployed. At the cold start stage, we use generic metrics to capture the similarity between a natural language query and a SQL template[1], and has to rely upon user feedback to make good choices. After collecting some

[1]The SQL templates can be manually generated by domain experts. In the cases when a history of SQL queries is available, the SQL templates can be automatically generated by analyzing the SQL history. In

expressions, our system maps the new query to the SQL template based on the previous expressions of that template. To collect the training data, we design an interactive communicator for the NLIDB. Highly ranked SQL templates with value instantiations are explained for the user to choose from. Once the user make a choice, NLQ paired with the chosen SQL template is collected, which serves as the prefect training set to improve the system. The choice action performed by the user serves as implicit verification of the interpretation. As such, we could assume that most of the training examples collected in this process are correct. In some extreme cases, a small part of the training examples collected might be incorrect matching pairs, which will disturb the mapping for the future similar queries. As will be discussed, our system prefers to rank a SQL template higher if it is associated with more previous NLQs showing high similarity with the new query. In such design, the desired SQL template are likely to be mapped, as long as most of the relevant training examples are correct.

Besides finding the desired template, the slots in the template must be instantiated correctly to generate the correct SQL statement. As such, the ambiguities at entity level need also to be fixed. The identification for an entity used in natural language scenarios and stored in a database are often different. In a database, keys are defined to distinguish entities from each other while in natural language scenarios, the identifications are often the text values. However, text values stored in the database may be very similar or even exactly the same, even if they are corresponding to different entities. Consider Example 13 again. There might be more than one author in the database named Feifei Li and have papers in SIGMOD. The system needs to figure out which one is the desired from the user. In our system, we model the database as a data graph. The nodes that are connected by many paths are considered strongly connected. When disambiguating the entities, we prefer to choose the combination that is strongly connected to each other, in which the context in both the NLQ and the data graph are taken into account.

Putting the ideas together, we propose a framework for building NLIDBs. The NLIDBs constructed could work even in the cases when no training data is available. It collects the training data through real usage and benefits from it accordingly. The intellectual contributions of this paper are as follows:

1. The problem of natural language interpretation is modeled as finding its corresponding SQL template and instantiating the template. We present a generic metric to evaluate the relevance between NLQ and SQL template, which could work even in the cases when no training examples are available.

database applications, a set of SQL queries is often cheaper to get than a set of (NLQ, SQL) pairs. As such, there are many cases when a SQL history is available but a log of (NLQ, SQL) pairs is not available.

2. We present metrics that are hyperparameter free, which could effectively detect the similarity between natural language expressions describing the same query logic, even in the cases when the training examples are limited.

3. We provide strategies for matching the entities mentioned in the NLQ to that stored in the database, in which the matching considers the context of both the NLQ and the data structure of the database.

4. Interactive communications are designed, in which multiple interpretations with explanations are returned for the user to choose from. This design could improve the recall and reliability of the NLIDB and most importantly, the user behavior data it collects could be used as training data to improve the performance of the system.

The remaining parts of the paper are organized as follows. In Section 4.2, we overview our strategy in disambiguation of natural language queries. Section 4.3 discusses the mapping from an NLQ to the SQL templates. We provide methods to resolve the ambiguities at entity level in Section 4.4. The system architecute is presented in Section 4.5. In Section 4.6, our system is evaluated experimentally. In Section 4.7, we draw conclusions.

4.2 Preliminaries

The input to our system is a natural language query whose semantic meaning may involve comparisons, aggregations, various types of joins and nestings, among other things. The semantic coverage of our system is a set of predefined SQL templates. Given a natural language query, by mapping it to the correct SQL template and instantiating the values, our system translates the natural language query into the desired SQL statement and evaluates it against an RDBMS. In this section, we first overview our solution and then define the SQL template.

4.2.1 Semantic Coverage

The problem we attempt to address is to translate a natural language query (NLQ) to a SQL statement. This problem would be difficult if it is modeled as a semantic parsing problem or machine translation problem, like most previous works did. The fundamental problem is that natural language queries are inherently ambiguous. Even a grammatically correct natural language sentence may contain ambiguities, from both the perspective of natural language processing and the perspective of database schema matching. Moreover, naive users often

use informal expressions and prefer brevity to logical precision. These challenges make it difficult to understand the query and compose the correct SQL statement from the scratch.

Example 14 *Consider a simple query "papers with 5000 citations" on the database shown in Figure 4.1. Many ambiguities exist. For example, in this query, "5000 citations" is more likely to mean "more than 5000 citations". Also, when generating the join path between "paper" and "citation", it is hard to figure out the direction of "citing" and "cited", in which one of them means citation while the other means reference. Due to these ambiguities, the query can correspond to four SQL statements with semantics that are different, leading to different query results.*

In our strategy, we would like to simplify the problem. First, given a database, we do not need to understand any natural language query beyond the scope of the information that can be provided by the database. Second, we only need to support the query logics that are semantically meaningful, which are only a very small subset of syntactically valid ones. Consider the Example 14 again. All the four interpretations are syntactically valid but only the correct interpretation out of the four interpretations are semantically meaningful, since users are not likely to query a paper with an exact number of citation or a paper with more than a number of references. By not supporting the unnecessary query logics, many ambiguities are removed naturally. Based on this observation, the task of interpreting a natural language query would be much easier if the system is designed to support only the necessary query logics that are likely to be frequently asked. As such, in our system, we would like to define the semantic coverage of an NLIDB as a set of necessary SQL templates, which can either be enumerated by experts or learned from a query log of SQL queries. Then the task of natural language interpretation is simplified to recognizing the desired SQL template and filling the slots in the template.

4.2.2 SQL Template

Two queries are said to have the same SQL template if they have the same SQL structure (SQL keywords, relations, attributes, aggregation functions, operators, nestings), even though constants in the query have different values. A SQL template is used to represent such SQL queries by replacing the specific values in the SQL statement with slots. In some cases, two SQL templates may have exactly the same query logic. In such cases, these SQL templates are merged, in which each original SQL template serves as an expression of the template and the one that can be evaluated most efficiently over an RDBMS is considered as the default expression of the template.

Example 15 *The SQL template corresponding to "authors who have more than 30 papers, in which each of these papers have more than 30 citations" is shown in Figure 4.3. This SQL template may have other expressions. For example, the part corresponding to the "citation of each paper" can be written as a subquery, which has the exactly the same query logic but evaluated less efficiently. As such, the expression shown in Figure 4.3 is used as the default expression for the SQL template.*

```
SELECT a.name, COUNT(p.pid)
FROM author a, writes w, publication p
WHERE a.aid = w.aid AND w.pid = p.pid AND p.citation_count > #NUM
GROUP BY a.aid
HAVING COUNT(p.pid) > #NUM
ORDER BY COUNT(p.pid) DESC
```

Figure 4.2: A Sample SQL Template.

So the question is how to map the input natural language query to the SQL templates. Search engines also deal with a mapping problem, which maps a set of keywords to documents, and receive great industry success. As we think about mapping from a natural language query to the SQL templates, it is worthwhile to draw inspiration from search engines, and to see how they do the ranking. At the early stage of search engines, the mapping is mainly based on metrics combined of relevance (e.g. using TF-IDF) and the importance of webpages (e.g. estimated using PageRank) [65]. After collecting training examples in the form of (keywords, page clicked), the ranking is improved with learning to rank frameworks [56]. In our system, we adopts this general design, we start from using metrics to evaluate the relevance between a natural language query and SQL templates. After training examples are collected, the mapping is improved accordingly.

Training Example. The training examples are in the form of pairs of (NLQ, SQL). In preprocessing, a SQL query is transformed to SQL template. Similarly, the natural language query is also preprocessed to a natural language template by replacing the values with slots. For example, the natural language query "citations for 'constructing an NLIDB' by others" is preprocessed to "citations for $publication by others".

4.3 Query Mapping

In this section, we interpret the whole query by mapping it to the SQL templates. Given a natural language query NLQ and a SQL template T, we measure the relevance between

them based on two intuitions. First, if the information specified in NLQ and T heavily overlap, it is likely that T is the desired SQL template of NLQ. This kind of relevance could be measured even in the cases when there are no training examples available. This intuition will be quantified in Section 4.3.1. Second, if a previous query NLQ' paired with T exist in the training set and the new coming NLQ shows high similarity with query NLQ', it is very likely that NLQ should be mapped to T. Measurements are provided to evaluate the similarity between the natural language queries in Section 4.3.2. In Section 4.3.3, we map the natural language query to the templates based on both the SQL expression and the training examples.

4.3.1 Information Overlap

After the tokenization, which will be discussed in detail in the next section, the natural language query is represented as a set of the mapped SQL components. Given the fact that trivial nodes are not considered, most of the information specified in *NLQ* should be contained by the SQL template T, if it is relevant to *NLQ*. Reciprocally, one would think most of the information in T should also be contained in *NLQ*. However, natural language queries tend to be brief and often leave out things that "can be understood". Furthermore, natural language queries typically do not include schema-specific constructs such as FK-PK join paths, which are often by-products of normalization. So, in the reciprocal direction, we expect that most of the *major* information in T should also be contained in *NLQ*, according to a definition of *major* that we provide in the next paragraph. Putting these ideas together, *NLQ* is considered relevant to T if (a) the information in *NLQ* is contained by T, and (b) the major information in T is specified in *NLQ*.

Example 16 *For the query of "authors focusing on database usability" on the database in Figure 4.1, its corresponding SQL template has a join path from "author" to "keyword" through "write" and "paper". In this SQL template, "author" is the element that are returned, "keyword" has a value constraint of "database usability", which means both of them should be mentioned by the user. In contrast, "write" and "paper" are in the middle of the join path, which is likely to be omitted in a natural language query.*

In our sytem, intuition (a) is captured by computing the ratio of $|Info(NLQ) \cap Info(T)|$ and $|Info(NLQ)|$, in which $Info(NLQ)$ is the set of mapped SQL components of the phrases in *NLQ* and $Info(T)$ is the set of the SQL components in T. This ratio measures how much proportion of the information specifies in NLQ are also specified in T.

Similarly, the intuition (b) is captured as the ratio of $|MInfo(NLQ) \cap MInfo(T)|$ and $|MInfo(T)|$, in which $MInfo(NLQ)$ is the set of nodes in *NLQ*, while $MInfo(T)$ is all the

major nodes in the SQL template T. The major nodes are the nodes whose information is important and unlikely to be omitted by the user. For example, a schema element that is returned or has a value constraint is considered as a major node, since it is not likely to be omitted by the user, while the schema element that serves only as a middle node in the join path is not considered as a major node, since it is often implicit in the user's query.

In our system, the information overlap between a natural language query and a SQL template is defined as the average of the two ratios mentioned above.

4.3.2 Semantic Distance between Queries

To evaluate the similarity between sentences, early works often represent sentences as bags of words/n-grams or strings, and use metrics like Jaccard Coefficient or edit distance to evaluate the similarity between them. The limitation is that in these methods, the semantic similarity between different words is not captured. Recently, a stream of researches attempt to solve this problem by learning a latent low-dimensional representation of sentences and work well in document matching [45]. However, in our situation, all the natural language queries are describing some query logic focusing on a narrow domain (the underlying database). In such situation, if trained from a generic corpus, the low-dimensional space may not be able to distinguish the natural language queries with slightly different query logics, in which natural language queries may too close to each other in the low-dimension space even in the cases when they correspond to different SQL template. The most straightforward way is to train the model using the data in the querying domain. However, in most situations, it is almost impossible to obtain large numbers of training examples over the domain of the specific querying database.

In this section we introduce new metrics for the distance between natural language sentences. The metrics we designed are hyper-parameter free, which means they can be used even in the cases when the training examples are very limited. Specifically, our approach leverages the recent results of word2vec [60], which embeds words to low-dimensional vectors, in which words semantically similarity to each other are close in the space. The authors demonstrate that semantic relationships are often preserved in vector operations on word vectors, such as vec(Picasso) - vec(painter) $\approx$ vec(Einstein) - vec(scientist) and vec(sushi) - vec(Japan) $\approx$ vec(bratwurst) - vec(Germany) [60]. Follows [43], our metrics utilize this property of word2vec embeddings to capture the distance between individual words. We represent each word as low-dimensional vectors and model sentences as sets of low-dimensional vectors. The distance between two sentences is defined as the minimum cost of traveling the words in one sentence to the words in the other sentence.

Example 17 *Consider a natural language query "authors with h-index over 40" (NLQ). It is very hard to map NLQ to the SQL template shown in Figure 4.3 directly. However, if we have a training example, which is "whose h-index is larger than 35" (NLQ') paired with the SQL template in Figuree 4.3, capturing the similarity between NLQ and NLQ' could help the system to understand NLQ. We adopt the basic idea in [43]. All words of both queries are embedded into a word2vec space. The distance between the two documents is the minimum cumulative distance that all words in NLQ need to travel to exactly match NLQ'.*

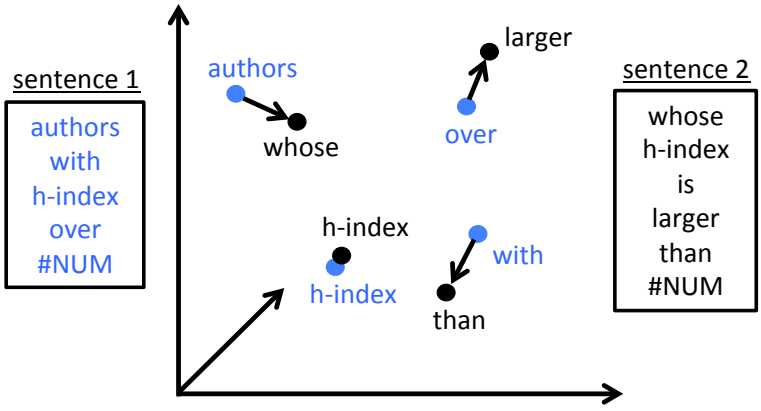


Figure 4.3: Word to Vector Embedding.

The major problem in the Word Mover's Distance provided in [43] is that common words and rare words have the same impact in the distance evaluation. Intuitively, the fact that a very rare word in one sentence cannot find similar words in the other sentence should cause a large distance for the two sentences. In contrast, if two sentences only differ in the semantic of some common words (or even stop words), they should not be assigned a large distance. As such, we take the term frequency-inverse document frequency (TF-IDF) of each word into account, in which the influence of rare words is augmented while influence of common words is reduced.

Word Travel Cost Our goal is to incorporate the semantic similarity between individual word pairs (e.g. coauthor and collaborator) into the distance metric between two query sentences. One such measure of word distance is naturally provided by their Euclidean distance in the word embedding space. More precisely, the semantic distance between word w and word w' becomes $||vec(w) - vec(w')||$. In addition, when traveling one word to another word, the cost is defined as $tfidf(w) * ||vec(w) - vec(w')||$.

Word Travel Cost from Sentence to Sentence The travel cost from one word to another is a natural building block to create a distance between two sentences. Let NLQ and NLQ' be two natural language sentences. The word travel cost from NLQ to NLQ' is defined as the minimum cumulative cost required to travel each word in NLQ to the words in NLQ'.

Sentence Travel Distance Given two natural language queries NLQ and NLQ', the sentence travel distance between them is defined as the average of the word travel cost from NLQ to NLQ' and the word travel cost from NLQ' to NLQ. We denote it as $Travel(NLQ, NLQ')$.

4.3.3 Template Mapping

As mentioned in Section 4.2.2, a query logic may be expressed in different SQL expressions. In such cases, the SQL templates with the same query logic are merged into one SQL template, in which each original SQL template serves as an SQL expression of the template. Also, given a set of training examples, there might be multiple natural language expressions corresponding to the same SQL template. As such, in our system, a SQL template is associated with m SQL expressions and n natural language expressions, in which m is at least 0 and n grows with the gathering of training examples for that template. Here we denote all the know expressions (including SQL expressions and natural language expressions) for the template T as $Expression(T)$.

When specified in natural language, the same query logic might have very different expressions. As such, when deciding whether a natural language query NLQ should be mapped to a SQL template T, the fact that NLQ shows high similarity with some expressions in $Expression(T)$ often means a lot. In contrast, the fact that NLQ shows low similarity with some expressions in $Expression(T)$ does not mean that they should not match. So, when evaluating the probability of matching NLQ with T, we focusing only on the old expressions showing high similarity with NLQ and see how many of them are coming from $Expression(T)$.

In our system, an natural language (resp. SQL) expression is considered to describe the same query logic with the natural language query NLQ when their distance (resp. similarity) is below (resp. above) a threshold. Based on this assumption, the probability that NLQ should be mapped to T can be computed as the ratio of the number of expressions in $Expression(T)$ showing high similarity with NLQ over the number of expressions of all the templates showing high similarity with NLQ.

Phrase Type	Corresponding SQL Component
Select Phrase	SQL keyword: SELECT
Operator Phrase	an operator, e.g. =, <=, !=, contains
Function Phrase	an aggregation function, e.g., AVG
Name Phrase	a relation name or attribute name
Value Phrase	a value under an attribute
Quantifier Phrase	ALL, ANY, EACH
Logic Phrase	AND, OR, NOT

Figure 4.4: Different Types of Phrases.

4.4 Entity Disambiguation

Given an online natural language query, before finding its corresponding SQL templates in the semantic coverage, we first interpret its words/phrases by mapping them to the database elements (SQL keywords, schema elements and database values). Such phrases can be further divided into different types as shown in Figure 4.4, according to the type of database elements they mapped to. The identification of select phrase, operator phrase, function phrase, quantifier phrase and logic phrase is independent of the database being queried. Following [51, 47], we enumerate sets of phrases as the database independent lexicon to identify these five types of phrases.

In this process, the major challenge is the matching between the entities mentioned in natural language query and that stored in the database. The problem comes from the fact that the identification for an entity in a natural language description and that stored in a database are often different. In natural language query, users often mention an entity by its names, while in databases, keys are defined to distinguish entities from each other. Often, the users are not able to specify the exact name of the entity that stored in the database (e.g. the title of a paper). Also, many different entities in the database may have similar or even exactly the same names (e.g. names of authors). As a result, approximate matching is necessary but the matching considering only spelling is not enough. The system must figure out which entity is the desired one, given the phrases specified by the user and that stored in the database.

We notice that there may be more than one entity specified in the natural language query. In that case, the fact that they are mentioned in one query means they are related to each other. When matching them with the entities in the database, we prefer to choose the mapped entities are also strongly "related" in the database. Also, we notice that the important entities are often more likely to be queried. So when we would also like to capture the importance of entities in the mapping.

Example 18 *Consider the query in Example 13. There are many authors in the database named Feifei Li and several conferences/journals have names contained SIGMOD. If one of the Feifei Li has many papers in one of the conferences/journals, it is very likely that this combination would be the desired one.*

In this section, we capture the intuitions of "related" and "importance" in a unified way. To facilitate that, the database is modeled as data graph.

Definition 12 (Data Graph) *The data graph is a undirected graph, $G = (V, E)$ with node set V and edge set E, where nodes in V are tuples in the database. There exists an edge between node v_1 and node v_2 if their corresponding relations have a foreign key to primary key relationship, and the foreign key of v_1 equals to the primary key of v_2.*

Example 19 *Consider the bibliographic database in Figure 4.1 again. The whole database is modeled as a data graph, in which each publication, author, organization, conference, journal, keyword and domain is modeled as a node. There are too many kinds of edges in the database and we take the edges related to a publication as an example. There is an edge between a publication and its authors, its published journal or conference, its keywords, its references and its citations.*

Definition 13 (Match) *Let k be an entity phrase specified in the natural language query, m be a tuple in the database, and $sim(k, m)$ be a similarity metric between k and m. Given a threshold τ, m is called a match of k when $sim(k, m) \geq \tau$. We use $M_k$to denote all the matches of k.*

Definition 14 (Match Combination) *Let $K = \{k_1, ..., k_n\}$ be the n entity phrases specified in the natural language query, M_{ki} be the set of all the matches of k_i, the set of match combination M_K is defined as $\{m_{k1}, ..., m_{kn} \mid m_{ki} \in M_{ki} \wedge 1 \leq i \leq n\}$ is called a match combination of K.*

In the query mentioned in Example 13, Feifei Li and SIGMOD are the two entity phrases. Suppose that the Professor Feifei Li in University of Utah and the Professor Fei-fei Li in Stanford University are the two matches of entity phrase Feifei Li, and the SIGMOD conference and SIGMOD Record are the two matches of the entity phrase SIGMOD. These two ambiguities would make up four match combinations.

Definition 15 (Joining Network of Tuples (JNT)) *Let $G(V, E)$ be the data graph, K be the set of entity phrases and m_K be a match combination of K. $J(V', E')$ is called a Joining Network of Tuples (JNT) of m_P when $J(V', E')$ is an acyclic subgraph of $G(V, E)$ and V' contains all the tuples in m_P, and each end node is in m_P.*

	#papers in SIGMOD Conference	#papers in SIGMOD Record
Feifei Li (Stanford)	1	0
Feifei Li (Utah)	12	2

Figure 4.5: Number of Subgraphs by each Entity Combination

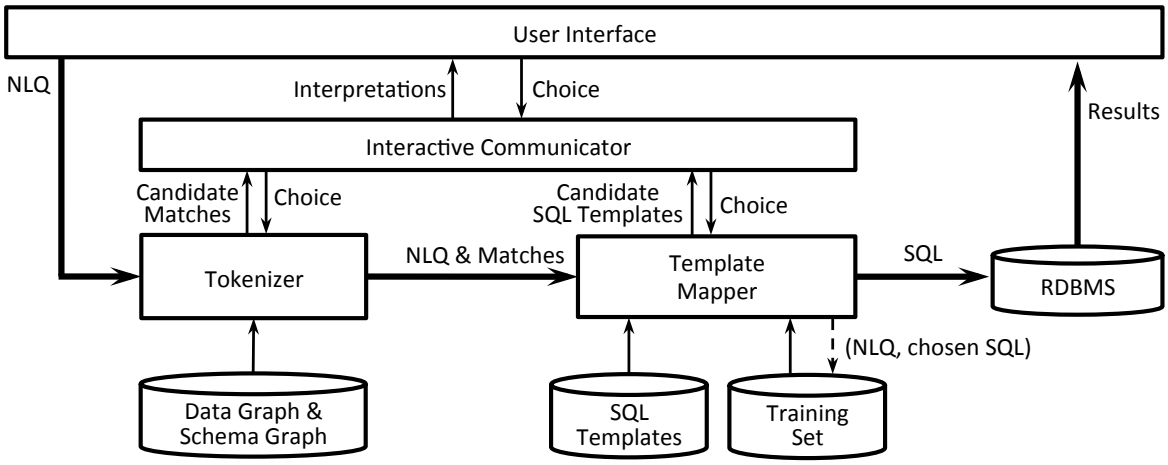


Figure 4.6: System Architecture.

Definition 16 (Combination Appearance) *Given a match combination M and a join path length threshold L, the size of the $JNTs$ of M within length L is defined as its match combination appearance. We denote it as A(M).*

Consider again the query in Example 13 and the database in Figure 4.1. If the length threshold for the join path is set as 2, then the number of papers by each match combination of Feifei Li and SIGMOD is the combination appearance.

Given the concept of combination appearance for each match combination, the probability that a combination M is the desired combination is computed as the ratio between the combination appearance of M and the total combination appearances of all the match combinations. We denote this ratio as $P(M)$.

Example 20 *Consider the Example 18 again. Assume that the number of papers by each Feifei Li in SIGMOD conference/Record is the number shown in Figure 4.5, in which each number is the combination appearance if we set the length threshold as 2. Then the probability that Feifei Li mentioned is the Professor in Utah and SIGMOD corresponds to SIGMOD Conference is 12/15 = 0.75. The probability of other combinations can be computed in the same way.*

The final overall matching score for a match combination is defined as follows, in which M is a match combination, in which an entity phrase k_i maps to m_{ki}.

$$P(M) * \sum_{i=1}^{n} sim(k_i, m_{ki})$$

When returning the results to the user, we choose the match combination with the highest connectiveness to the user as the default match for all the entities. Also, for interactive NLIDBs, it is often useful to return the user alternatives to choose from [47]. In such case, candidate matches need to be generated individually for each single entity.

Given the concept of combination appearance, we take a step further to compute the total appearance for each match as the total combination appearance of the combinations that contains the match. The probability of each match to be the desired one is defined as the ratio between its appearance and the total appearance of all the match combinations. We denote the probability as $P(m)$, in which m is the match.

Example 21 *Consider Example 21 again. The appearance of the match from Feifei Li to the professor in Utah is (12+2) = 14 and the probability of this match is 14/15. The probability of other matches can be computed in the same way.*

The overall matching score for a match is defined as $P(m) * sim(k, m)$. For each individual entity, we rank its matches based on the overall matching score and returns the top ones for the users to choose from.

4.5 System Architecture

Figure 4.6 depicts the architecture of our system. The entire system consists of three main parts: the tokenizer, template mapper, and interactive communicator. The tokenizer is responsible for resolving the ambiguities at the words/phrases level, while the template mapper is responsible for returning the top ranked SQL templates for the natural language query. The interactive communicator is responsible for communicating with the user to ensure that the interpretation process is correct. It also collects the training data, in which in the form of (NLQ, SQL) for future use, after the verification of the user.

Tokenizer The tokenizer identifies the sets of nodes in query that can be mapped to database elements (SQL keyword, schema elements and values in the database). In this process, a set of words may be merged into a phrase, which should be taken as a whole to

form a single token. The major challenge in this part is that the identification for an entity in natural language scenarios is often text values, while in the database, an id is often defined to distinguish entities from each other. In the database, some text values may similar to each other or even be exactly the same, and the users may not able specify the exact text values. As such the identification of entities are often a problem. In the tokenizer of our system, we prefer to choose the matches that are strongly connected with each other in the data graph, in the cases when multiple matches exist.

Template Mapper The next step is to understand the whole query. In the template mapper, a natural language query is interpreted by finding its corresponding SQL template. Specifically, our system provides the metric to evaluate the relevance between a natural language query and the SQL templates. This metric serves as the major evidence for the mapping in the cases when no training data is available. After the system obtains the training examples through the usage, the mapping is improved by considering the relevance between the new query and previous queries.

Interface for End Users To deal with the possibility that the system may misunderstand the user, our system explains to the user how her query is processed. Specifically, the interactive communicator returns multiple choices with explanations for the user to choose from. Since the choices are explained, the chosen behavior of the user implies the fact that this interpretation has been verified. As such, the user behavior in this process, which is in the form of NLQ and SQL pairs, serves as the prefect training set to improve the system. This design enables our system to improve itself through real usage.

4.6 Experiments

Given an online natural language query, we first interpret each of its entities by mapping it to the database values. Then, the whole query is interpreted by finding its corresponding SQL templates from the semantic coverage. As such, we evaluate the entity disambiguation and the template mapping in Section 4.6.1 and 4.6.2, respectively.

Dataset. The dataset we use is Microsoft Academic Search (MAS), whose simplified schema graph is shown in Figure 4.1. Some summary statistics of its major relations are shown in Figure 4.7.

Relation	#tuples	Relation	#tuples
Publication	2.45 M	Author	1.25 M
cite	20.3 M	writes	6.02 M
Conference	2.9 K	Journal	1.1 K
Organizations	11 K	Keywords	37 K

Figure 4.7: Summary Statistics for MAS Dataset.

4.6.1 Disambiguation at Entity Level

As mentioned, when mapping the phrases specified by the user to the entities stored in the database, multiple matches might exist and the system need to figure out which ones are the desired. In our system, two tasks exist. First, we need to find the best combination of matches for the set of phrases specified in a query. All matches in this combination are used as the default mapping for each entity. Second, our system is designed as an interactive system, which returns multiple mappings for each ambiguous entity. As such, we need to generate a ranked list of matches as alternatives for the user to choose from, which aims to do the entity matching correctly under the help of the user, even in the cases when the default combination of matches is wrong. Specifically, we generate 5 alternative matches for each entity phrase.

Comparisons For the task of generating the best match combination, most previous works in graph-based keyword search [9, 25, 35] just take the match combination in the shortest joining network of tuples. Our method takes a step further, which considers the combination appears most frequently in all the join networks of tuples within a length threshold. In the experiments, we compare our strategy with that provided in [35], in which the match combination in the shortest (with lowest weight) joining network of tuples. For the task of generating alternative matches for each single entity phrase, previous works often rank the matches based only on the spelling [47]. In contrast, we take both the spelling and the appearance frequency of each single match into account. We compare our strategy with that provided in [47] in experiments.

Test Set Unfortunately, MAS does not make its query history public. Therefore, we created our own test set from 5 PhD students majoring in computer science or information science. All of them have the need to do academic searches and are familiar with many of the famous authors, publications, conferences, journals, organizations, organizations, and hot topics in the subdomain they are focused. Each of them was asked to specify 5 groups of ambiguous

1. Papers by Peter Chen in Database area
2. Paper titled like "keyword search in databases" by authors in CUHK
3. Author named Mark Newman working on Social Network

Figure 4.8: Samples of Groups of Entities.

entities. In a group, each entity are often not easy to be distinguished by itself but can be distinguished when all the entities in the group are considered. Sample examples are shown in Figure 4.8. Thus we obtained a test set comprising 20 groups of entities. The number of entities mentioned in these 20 groups is 42.

Results The experimental results for entity disambiguation is shown in Figure 4.9. For the task of generating the best match combination, the strategy of choosing the match combination in the shortest (with lowest weight) joining network of tuples achieves only a precision of 11/20. In this strategy, the information in the shortest joining network serves as the only evidence for the matching, which is not often enough to figure out the desired match accurately. For example, in the first sample group of entities in Figure 4.8, the phrase "Peter Chen" has multiple matches like the Professor who developed the ER model and the Professor in University of Michigan focusing on operating systems and distributed systems. Both of the two professors have publications in the database area, which makes it hard to choose the correct one using previous approaches[2]. In contrast, when we do the entity disambiguation, we take all the joining networks of tuples into account. For the first sample group of entities, the Professor who developed the ER model has a much higher combination appearance with the other entity phrase "database". As such, our system ranks this combination higher.

For the individual ranking, our system outperforms the previous strategy significantly. The reason is that the entities in the test set are all ambiguous ones, in which most of them have many matches that are very similar or even exactly the same with each other. Also, the specifications of the entity phrases are not often exactly the same with their corresponding entities stored in the database. As such, the spelling similarity itself is not enough to provide a good ranking. In our strategy, the relationships between different entities are considered to do the disambiguation, even in the cases when we rank the matches for each single entity

[2]Most graph-based keyword search approaches [9, 25, 35] believe that an edge would carry more information if there are less edges emanated from the node containing the foreign key. As such, when generating the joining networks of tuples, they prefer to choose the joining network comprising with the nodes having less connections. However, from our experience and experiments, users are often querying the entities with high connections (famous authors, publications, or conferences). That would make these approaches work even worse than randomly picking one.

	Previous Works	New Approach
Match Combination	11/20	18/20
Alternative Matches	26/42	42/42

Figure 4.9: Entity Disambiguation.

phrase. That helps our system to provide the alternatives accurately in the entity disambiguation process.

4.6.2 Template Mapping

In our system, the primary step in interpreting a natural language query is mapping it to its desired SQL template. Specifically, the mapping can be done when no training examples are available and can be improved when training examples are collected. In this section, we test the effectiveness in the mapping and show the benefits brought from the training examples.

Query Task Design To test the effectiveness of the mapping process, we need a set of query tasks. Ideally, these query tasks should cover most of the query logics that are possible to be asked over the database. To obtain such a set of query tasks, we examine the old version interface of MAS dataset. It was a carefully designed faceted interface. By clicking through the site, a user is able to get answers to many quite complex queries. We enumerated all query logics that are directly supported by the MAS website and obtained a set of 196 SQL templates. We noticed that the semantic coverage of the MAS website was constrained by the overall design of the interface. As a result, some likely query logics are not supported because they would require a redesign of the interface and impact the overall visual effect. Also, it is possible that some unnecessary query logics are supported just because they are more compatible with interface. To address these shortcomings, we asked five domain experts, who often do academic searches and have a good sense of which questions would be frequently asked over a bibliographic database, to modify this set of query logics, deleting unnecessary ones and adding new ones as needed. In this process, we accepted a modification only when it had the agreement of all five experts. The final set of query tasks Q_{gold} we obtained is composed of 160 different query logics (corresponds to different SQL templates). Specifically, Q_{gold} contains various kinds of complex queries, in which 110, 52, 53, 34 of their corresponding SQL templates contain aggregations, long join paths with length ≥ 5, subqueries, and multilevel of subqueries, respectively.

Training Set and Testing Set The motivation of our system is to enable users to query the database though natural language. As such, the training set and testing set must be collected from real users. In the collection process, sixteen participants were recruited with flyers posted on a university campus. A questionnaire indicated that all participants were familiar with keyword search interfaces (e.g. Google), but had little knowledge of formal query languages (e.g. SQL). Furthermore, they were fluent in both English and Chinese. For our experiments, it is hard to convey the query task to the participants since any English description would cause bias in the task. To overcome this, we described each query task in Chinese and asked users to compose English query sentences. Since English and Chinese are in entirely different language families and we believe this kind of design can minimize such bias. We transformed each of the query logics into a query task and evenly divided the query tasks into 16 task groups, in which each task group contained 10 query tasks. Each participant randomly took 5 task groups and composed the natural language query according to the Chinese description. After that, we got 800 natural language queries, in which each query logic in Q_{gold} obtained 5 natural language queries.

For each experiment, we randomly pick one natural language query for each query logic in Q_{gold}. These 160 natural language queries form the testing set. A subset of the other 640 natural language queries together with their corresponding SQL templates are used as the training set, according to the number of training examples needed.

Semantic Coverage In our system, the quality of the semantic coverage directly affects the behavior of the online part. In the experiments, a set of 300 SQL templates are adopted as the semantic coverage, which is automatically generated and contains all the 160 query logics in Q_{gold}.

Measurements We evaluate effectiveness of the top 1 mapping as $|M_P|/|M_T|$, in which $|M_T|$ is the total number of queries and $|M_P|$ is the number of queries that can be mapped to the correct SQL templates in the top 1 mapping. Remember that our system also returns alternative SQL templates for the user to choose from. As such, we also evaluate the effectiveness of the top 5 mappings returned as $|M_R|/|M_T|$, in which M_R is the number of queries in which one of the top mappings returned is correct. Specifically, our system returns the user at most five candidate interpretations.

Results The results are shown in Figure 4.10. $|M_P|$ grows from 98 to 121 when training examples are added. $|M_R|$ grows from 134 to 156 with the training examples. We also compare our system with NaLIR [47] and TBNaLIR. NaLIR is a generic NLIDB whose

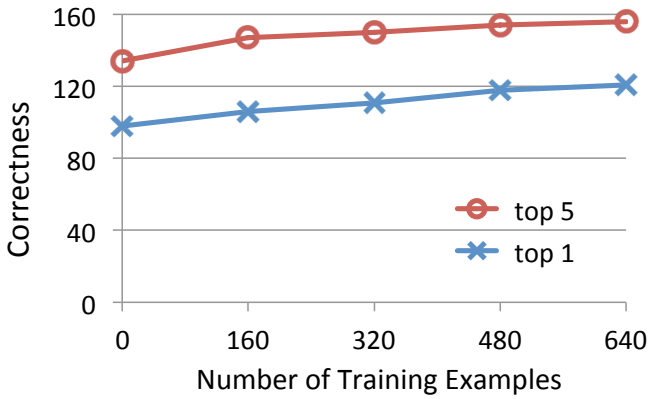


Figure 4.10: Results for Template Mapping.

semantic coverage is the syntactically valid SQL statements (under some syntactical constraints). TBNaLIR is a template-based NLIDB, which maps natural language query to the weighted SQL templates using generic metrics. Specifically, the SQL templates used in TBNaLIR are the same as that used in our system. Using NaLIR, the $|M_P|$ and $|M_R|$ are 84 and 101, respectively. For TBNaLIR, the $|M_P|$ and $|M_R|$ are 101 and 135, respectively. Our method outperforms NaLIR and TBNaLIR a lot when training examples are added.

4.7 Conclusion

In this paper, we provide a framework for constructing natural language query interfaces over relational databases. In the framework, the semantic coverage of an NLIDB is defined as a set of SQL templates. Given a natural language query, by mapping it to the correct SQL template in the semantic coverage, the query can be translated into the desired SQL statement, which may include comparison predicates, conjunctions, quantifications, multi-level aggregations, nestings, and various types of joins, among other things. In the cases when no training data is available, the mapping is mainly based on the relevance metrics between the natural language query and SQL templates. When some training examples are obtained, learning to rank is adopted to improve the mapping. Our system also provide the interactions with the user, which collects the user behavior data as the training set and improve the behavior of our system through the usage.

CHAPTER 5

Related Work

5.1 Semantic Parsing

The problem of constructing Natural Language Interfaces to DataBases (NLIDB) has been studied for several decades. Early systems often model the problem as a semantic parsing problem, in which grammars are either manually specified [5] or learned from training examples [94, 78, 33, 80]. While quite successful in some specific scenario, the grammars are hard to scale up, both to other domains and to new natural language expressions, which limits the impact [55].

In recent years, deep learning methods have achieved great success in machine translation [29] and question answer matching [77, 32]. This fact inspires researchers to apply end-to-end frameworks to build NLIDBs [26, 57, 54, 70]. From our experience, three major challenges exist in adopting end-to-end framework to construct NLIDBs. First, the underlying database schema of different NLIDBs often differs, which means the training set used for one NLIDB cannot be applied to another. As a result, in real applications, it is almost impossible to collect enough pairs of (NLQ, SQL) for an end-to-end model for the querying database. Second, as pointed out in [69, 47], reliability is very important in database applications. Users will be very frustrated if they make wrong decisions only because they believe in the wrong results from an NLIDB. As such, explanations are often necessary for users to understand the processing process. However, for deep learning methods, the intermediate structures are often unexplainable. Third, as discussed in this dissertation, NLIDB has many unique resources for resolving ambiguities, like the underlying schema structure, the query log, the distribution of the data, the configuration from the DBA, and so forth. However, in end-to-end methods, the fact that more resources taken into account often means higher dimensions of input, which would exacerbate the first problem of lacking training examples. Considering the above obstacles, we develop a new framework, which model the natural language query interpretation problem as a mapping problem, from the query to an

interpretation in the semantic coverage instead of adopting the semantic parsing framework.

5.2 Generic NLIDB

One branch of research works focused on building generic NLIDBs [69, 68, 51, 47] based on schema mapping techniques and domain-independent grammars covering only the natural language expressions describing database queries. PRECISE [69, 68] defines a subset of natural language queries as semantically tractable queries and precisely translates these queries into corresponding SQL queries. NaLIX [50] defines a domain-independent grammar for the natural language queries that are semantically expressible by XQuery and parses natural language queries to XQuery statements according to the grammar. NaLIR [47] is designed to support complex natural language queries, in which the semantic coverage is defined as all the syntactically valid SQL statements (with some constraints). By transforming natural language queries to the correct points in the semantic coverage in a series steps, the queries can be translated to the desired SQL statements. In general, existing generic NLIDBs still define their semantic coverage at the syntax level. In our system, an offline part is used to refine the semantic coverage, which supports only the query logics that are likely to be queried. By greatly narrowing down the semantic coverage, the burden in online query interpretation is reduced.

5.3 Question Answer Systems on Knowledge Base

Compared with NLIDBs, the questions answered by QA systems on knowledge base are often board but shallow, like "who is Obama's wife?" and "when Obama got married?". Most traditional works on QA-KB fall in one of the three categories: semantic parsing, information extraction and vector modeling. Semantic parsing, as discussed, generates representations that are semantically meaningful to computers for the questions over knowledge base [7, 15, 44, 30]. Information extraction, when applied on QA over KB, extracts the entities and relationships in the questions. The extracted entities and relationships are then mapped to the subgraphs in the knowledge base to obtain the answers [87]. Vector modeling learns the mapping strategies between the distributed embeddings for the questions and the distributed embeddings for the answers [11, 86, 13].

Later, researchers apply deep learning models to QA over knowledge base and achieve great achievements. CNN is applied to vector modeling [27] and semantic parsing [89]. CNN and LSTM is adopted in information extraction, especially entity and relationship classification [85, 96, 95]. Recently, Memory Networks and Attention Mechanism is used to

answering questions over knowledge base [12, 98].

From our understanding, the task of understanding a natural language query over a database can be divided into two parts, understanding the question and understanding the database. For QA-KB, the questions are often simple but the topics of the underlying knowledge are very board. In this case, the focus should be on understanding the question since the understanding of the question itself is relatively easier, which is exactly the strategy adopted by most of the QA-KB works. In contrast, for relational databases, the questions are often complex while the topics of the underlying database are often very focused. In such case, the frequently asked questions are actually predictable. So we focus on understanding the database first, generating the semantic coverage that covers only the semantically meaningful query logics.

5.4 Keyword Search over Relational Databases

Keyword search interfaces are widely used by non-experts to specify ad-hoc queries over relational databases [93]. Two main families of methods are used: schema-based approaches (e.g. DISCOVER [37]) and graph-based approaches (e.g. BANKS [9]). Schema-based approaches first translate the keywords into a set of minimal total join networks (MTJN) and then evaluate them. In graph-based approaches, the database is modeled as a data graph, in which each node is a tuple [9]. Steiner trees of tuples that contain all the keywords are constructed directly on the data graph and outputted as results. In both of the two methods, keyword search is used to search joined entities for the input keywords, not answering questions directly.

Recently, there is a stream of such research on keyword search [79, 74, 17, 84, 31, 10, 8], in which, given a set of keywords, instead of finding the data relevant to these keywords, they try to interpret the query intent behind the keywords in the view of a formal query language. In particular, some of them extend keywords by supporting aggregation functions [79], Boolean operators [74], query language fragments [10], and so forth. These works can be considered as a first step toward addressing the challenge of natural language querying. Our work builds upon this stream of research and supports a richer query mechanism that allows us to convey much more complex semantic meaning than a flat collection of keywords.

5.5 Other Kinds of User Friendly Interfaces for Databases

5.5.1 Form-based Search

Form-based search is widely used to support fixed query logics. It is especially useful in the cases when the queries are focused and predictable. Most of the research works on form-based search focus on how to simplify the process of manually designing forms, while others try to provide semi-automatic or even automatic strategy for form construction. FoXQ [4] is a system that helps users build queries incrementally by navigating through layers of forms, and view results in the same way. EquiX [18] helps users build queries in steps using a form. These systems provide form-developers visual tools to generate forms manually, but the task is significantly simpler than traditional form-building tools. Semi-automatic form generation tools are presented in QURSED [66, 67, 16]. Later, Musiq [38, 39, 40, 41] is proposed to create forms in a pure automatic manner.

5.5.2 Faceted Search

Faceted search was originally introduced for browsing image and text collections [28, 88, 20, 21, 76]. Recently, there have been efforts on creating a faceted search interface for structured data [72, 23, 73]. Although the basic faceted interface is a very simple browsing model, the usability of faceted interface depends highly on the way the facets are chosen or the way search results are presented [75]. There are many interesting research areas in faceted search, such as identifying interesting facets and facet values [72, 23, 73], automatic construction of faceted interface [20, 76, 46], discovering OLAP type of interesting information [81, 6, 23]. Compared with other types of interfaces, faceted search is very useful in the cases when the query logics need to support is predictable. But it cannot support ad-hoc queries. Also, the query logics can be supported are often restricted by the overall design of the interface.

5.5.3 Visual Query Builders

Many visual query interfaces have been proposed to assist users in building queries incrementally through visual interfaces by constructing mappings between actions in the user interface and an existing restricted database language. [1, 2, 3] are widely used visual query builders that can be plugged on an RDBMS. VQL [62] addresses the design issues involved with querying complex schema. It introduces graphical primitives that map to textual query notations, allowing users to specify a query-path across multiple tables, and also express recursive queries visually. The Polaris System [36] constructs a similar mapping between

common user interface actions such as zooming and panning, and relational algebra. This allows users to visualize any standard SQL database using a point and click interface. Generally speaking, visual query builders are very effective to help experts to fast compose standard query languages, but they often still require users to understand the schema structure of the database, which means they are not easy to use for naive users.

5.5.4 Query by Examples

Query-by-Example (QBE) [100, 101] is a well-known work that allowed users to query a database by creating example tables in the query interface. The table is then translated into a more structured language, such as SQL, under both the database schema and the result view. Examples with same value suggest how the relations are joined and which attributes are projected. While being a friendlier approach to database querying than SQL, QBE does not perform well with large schema / data scenarios. Furthermore, the user is expected to be aware of the values of the database prior to the query evaluation.

5.5.5 Schema Free SQL/XQuery

The basic idea of schema-free SQL/XQuery is to allow users to specify query logics in SQL/XQuery without knowing the schema of the underlying database [52, 53, 49]. The users can specify queries based on whatever partial knowledge of the schema they have. If they know the full schema, they can write full SQL/XQuery. If they do not know the schema at all, they can just specify labeled keywords. Most importantly, they can specify queries somewhere in between. The system will respect whatever specifications are given and generate the formal queries.

5.5.6 Schema Summarization

Another approach to solve the complexity of schema is that of schema summarization [90]. The idea is to develop a representation of the underlying complex schema at different levels of detail. A user unfamiliar with the database would first be shown a high-level schema summary comprising only a small number of concepts. [91] allows a user to query the database through the schema summary directly, without the knowledge of the underlying complex schema and with high result quality and query performance.

5.6 Search Engines

The framework of the approach in this dissertation is inspired from search engines [14, 65]. Similar to NLIDBs, search engines are heuristic query systems, whose queries do not have formally defined semantic meanings. The semantic coverage of a search engine is a large set of webpages. In the early stage of search engines, search engines maps the keywords to the webpages based on predefined metrics, like the importance of each webpage (e.g. PageRank) and the relevance between the keywords and each webpage (e.g. tf-idf). After collecting training data, which is in the form of (keyword, webpage clicked), learning to rank frameworks are adopted to improve the quality of the mapping [56]. Inspired from search engines, we model the natural language query interpretation problem as a mapping problem between the natural language query to a point in the semantic coverage. In this framework, when no training examples are available, predefined metrics are provided to map the input query to the SQL templates. It also collects training data through the usage and benefits from the training data collected to improve its performance.

5.7 Word-embedding & Sentence Embedding

One problem in our system is how to evaluate the similarity between sentences. A stream of research attempts to solve this problem by learning a latent low-dimensional representation of sentences and work well in document matching [45]. However, this strategy does not work well in our situation, since all the questions over an NLIDB are describing some query logic focusing on a narrow domain (the underlying database). In such situation, if trained from a generic corpus, the low-dimensional space is not able to distinguish the natural language queries with slightly different query logics. If trying to train the system from the specific corpus of the NLIDB, collecting enough training examples is still a problem. So we adopt the general strategy provided in [43], which only does the embedding at the word/phrase level. The method we adopted is word2vec [59]. Their model learns a vector representation for each word using a (shallow) neural network language model. The authors demonstrate that semantic relationships are often preserved in vector operations on word vectors, such as vec(Picasso) - vec(painter) $\approx$ vec(Einstein) - vec(scientist) [60]. Learning the word embedding is entirely unsupervised and it can be computed on the text corpus of interest or be pre-computed in advance.

5.8 Entity Matching

The disambiguation at entity level is also a challenge in our system. One of its similar problem, entity matching, is investigated in the area of data integration and data cleaning [34]. Often, manually crafted or learned rules are applied to detect the entities describing the same real world entity between structured data. In our system, the entity matching is from the entities mentioned in natural language query and the entities stored in the database. Similar problems are dealt with in graph-based keyword search [9, 25, 35], in which the mapped entities in the shortest joining network of tuples are often considered as the best mapping. In contrast, we provide a principled strategy to compute the probability for each matching. Both intuitions and experimental results show the improvements over that used in graph-based keyword search.

5.9 Query Explanation

A task in our system is to explain the SQL templates for the end users to choose from or for the DBA to fast review. Previous systems explain SQL queries to users using natural language descriptions [42] or query visualizations [22, 31, 8]. In our system, we adopt the strategy used in [42].

CHAPTER 6

Conclusions and Future Work

Querying data in relational databases is often challenging since SQL is too difficult for naive users without technical training. Theoretically, an NLIDB would enable naive users to specify complex, ad-hoc query intent without training. This dissertation studies different issues involved in constructing NLIDBs.

Chapter 2 describes an interactive natural language query interface for relational databases. Given a natural language query, our system first translates it to a SQL statement and then evaluates it against an RDBMS. To achieve high reliability, our system explains to the user how her query is actually processed. When ambiguities exist, for each ambiguity, our system generates multiple likely interpretations for the user to choose from, which resolves ambiguities interactively with the user. The query mechanism described in this chapter has been implemented, and actual user experience gathered. Using our system, even naive users are able to accomplish logically complex query tasks, in which the target SQL statements include comparison predicates, conjunctions, quantifications, multi-level aggregations, nestings, and various types of joins, among other things.

In Chapter 3, we provide a framework for constructing a template-based natural language query interfaces for relational databases. In the framework, the semantic coverage of an NLIDB is defined as a set of weighted SQL templates, in which the weight describes the likelihood of a SQL template to be queried. Given a natural language query, by mapping it to the correct SQL template in the semantic coverage, the query can be translated into the desired SQL statement, which may include comparison predicates, conjunctions, quantifications, multi-level aggregations, nestings, and various types of joins, among other things. We provide the principled strategy for automatically generating the semantic coverage from the query log, according to the random user model we assumed. Also, an effective mapping strategy, which considers the template weights, as well as the relevance between the query and the templates, is proposed. The framework described in this chapter has been implemented, and actual user experience gathered. Using our system, a small sized query log is

enough to generate the necessary SQL templates, and even naive users are able to accomplish logically complex query tasks against our system.

Chapter 4 describe a framework of constructing NLIDBs that can be improved through the usage. Following the previous work, the query interpretation process is modeled as the mapping from the natural language query to the SQL templates in the semantic coverage. At the cold start stage when no training examples are available, the mapping is mainly based on generic metrics without hyperparameters. When some training examples are obtained, learning to rank is adopted to improve the mapping. Our system also provide the interactions with the user, which collects the user behavior data as the training set and improve the behavior of our system through the usage. Using our strategy, an NLIDB would work at the very beginning, collect training data through real usage, and improve itself with training data is gathered.

In this dissertation, the process of natural language queries is independent of each other. Actually, search is not often a single-step process. A user may ask follow-up questions based on the results obtained. It is thus necessary to provide a system to support a sequence of related queries. In the future, we would like to explore how to support follow-up queries, thereby allowing users to incrementally focus their query on the information they are interested in, especially in conversation-like interactions.

BIBLIOGRAPHY

[1] Active query builder: www.activequerybuilder.com.

[2] Flyspeed sql query: www.activedbsoft.com.

[3] Sqleo visual query builder: sqleo.sourceforge.net.

[4] R. Abraham. Foxq -xquery by forms. In *2003 IEEE Symposium on Human Centric Computing Languages and Environments (HCC 2003), 28-31 October 2003, Auckland, New Zealand*, pages 289–290, 2003.

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

[6] O. Ben-Yitzhak, N. Golbandi, N. Har'El, R. Lempel, A. Neumann, S. Ofek-Koifman, D. Sheinwald, E. J. Shekita, B. Sznajder, and S. Yogev. Beyond basic faceted search. In *Proceedings of the International Conference on Web Search and Web Data Mining, WSDM 2008, Palo Alto, California, USA, February 11-12, 2008*, pages 33–44, 2008.

[7] 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.

[8] S. Bergamaschi, F. Guerra, M. Interlandi, R. T. Lado, and Y. Velegrakis. Quest: A keyword search system for relational data based on semantic and machine learning techniques. *PVLDB*, 6(12):1222–1225, 2013.

[9] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, 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] A. Bordes, S. Chopra, and J. Weston. Question answering with subgraph embeddings. 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 615–620, 2014.

[12] A. Bordes, N. Usunier, S. Chopra, and J. Weston. Large-scale simple question answering with memory networks. *CoRR*, abs/1506.02075, 2015.

[13] A. Bordes, J. Weston, and N. Usunier. Open question answering with weakly supervised embedding models. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2014, Nancy, France, September 15-19, 2014. Proceedings, Part I*, pages 165–180, 2014.

[14] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[15] Q. Cai and A. Yates. Large-scale semantic parsing via schema matching and lexicon extension. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, 4-9 August 2013, Sofia, Bulgaria, Volume 1: Long Papers*, pages 423–433, 2013.

[16] B. Cautis, A. Deutsch, N. Onose, and V. Vassalos. Efficient rewriting of xpath queries using query set specifications. *PVLDB*, 2(1):301–312, 2009.

[17] E. Chu, A. Baid, X. Chai, A. Doan, and J. F. Naughton. Combining keyword search and forms for ad hoc querying of databases. In *Proceedings of the ACM SIGMOD*, pages 349–360, 2009.

[18] S. Cohen, Y. Kanza, Y. A. Kogan, Y. Sagiv, W. Nutt, and A. Serebrenik. Equix - A search and query language for XML. *JASIST*, 53(6):454–466, 2002.

[19] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158, 1971.

[20] W. Dakka, P. G. Ipeirotis, and K. R. Wood. Automatic construction of multifaceted browsing interfaces. In *Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management, Bremen, Germany, October 31 - November 5, 2005*, pages 768–775, 2005.

[21] W. Dakka, P. G. Ipeirotis, and K. R. Wood. Faceted browsing over large databases of text-annotated objects. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 1489–1490, 2007.

[22] J. Danaparamita and W. Gatterbauer. Queryviz: helping users understand sql queries and their patterns. In *EDBT*, pages 558–561, 2011.

[23] D. Dash, J. Rao, N. Megiddo, A. Ailamaki, and G. M. Lohman. Dynamic faceted search for discovery-driven analysis. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management, CIKM 2008, Napa Valley, California, USA, October 26-30, 2008*, pages 3–12, 2008.

[24] M.-C. de Marneffe, B. MacCartney, and C. D. Manning. Generating typed dependency parses from phrase structure parses. In *LREC*, pages 449–454, 2006.

[25] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, 2007.

[26] L. Dong and M. Lapata. Language to logical form with neural attention. In *ACL*, 2016.

[27] L. Dong, F. Wei, M. Zhou, and K. Xu. Question answering over freebase with multi-column convolutional neural networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, pages 260–269, 2015.

[28] J. English, M. A. Hearst, R. R. Sinha, K. Swearingen, and K. Yee. Hierarchical faceted metadata in site search interfaces. In *Extended abstracts of the 2002 Conference on Human Factors in Computing Systems, CHI 2002, Minneapolis, Minnesota, USA, April 20-25, 2002*, pages 628–639, 2002.

[29] Y. W. et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[30] A. Fader, L. Zettlemoyer, and O. Etzioni. Open question answering over curated and extracted knowledge bases. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, pages 1156–1165, 2014.

[31] J. Fan, G. Li, and L. Zhou. Interactive SQL query suggestion: Making databases user-friendly. In *ICDE*, pages 351–362, 2011.

[32] M. Feng, B. Xiang, and B. Zhou. Distributed deep learning for question answering. In *CIKM*, pages 2413–2416, 2016.

[33] R. Ge and R. Mooney. A statistical semantic parser that integrates syntax and semantics. In *CoNLL*, pages 9–16, 2005.

[34] L. Getoor and A. Machanavajjhala. Entity resolution: Theory, practice & open challenges. *PVLDB*, 5(12):2018–2019, 2012.

[35] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD*, pages 927–940, 2008.

[36] P. Hanrahan. Vizql: a language for query, analysis and visualization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, page 721, 2006.

[37] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[38] M. Jayapandian and H. V. Jagadish. Automating the design and construction of query forms. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 125, 2006.

[39] M. Jayapandian and H. V. Jagadish. Automated creation of a forms-based database query interface. *PVLDB*, 1(1):695–709, 2008.

[40] M. Jayapandian and H. V. Jagadish. Expressive query specification through form customization. In *EDBT 2008, 11th International Conference on Extending Database Technology, Nantes, France, March 25-29, 2008, Proceedings*, pages 416–427, 2008.

[41] M. Jayapandian and H. V. Jagadish. Automating the design and construction of query forms. *IEEE Trans. Knowl. Data Eng.*, 21(10):1389–1402, 2009.

[42] A. Kokkalis, P. Vagenas, A. Zervakis, A. Simitsis, G. Koutrika, and Y. E. Ioannidis. Logos: a system for translating queries into narratives. In *SIGMOD Conference*, pages 673–676, 2012.

[43] M. J. Kusner, J. R. Gardner, R. Garnett, and K. Q. Weinberger. Differentially private bayesian optimization. In *ICML*, pages 918–927, 2015.

[44] T. Kwiatkowski, E. Choi, Y. Artzi, and L. S. Zettlemoyer. Scaling semantic parsers with on-the-fly ontology matching. 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 1545–1556, 2013.

[45] Q. V. Le and T. Mikolov. Distributed representations of sentences and documents. In *ICML*, volume 32 of *JMLR Workshop and Conference Proceedings*, pages 1188–1196. JMLR.org, 2014.

[46] C. Li, N. Yan, S. B. Roy, L. Lisham, and G. Das. Facetedpedia: dynamic generation of query-dependent faceted interfaces for wikipedia. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 651–660, 2010.

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

[48] F. Li, T. Pan, and H. V. Jagadish. Schema-free sql. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1051–1062, New York, NY, USA, 2014. ACM.

[49] F. Li, T. Pan, and H. V. Jagadish. Schema-free SQL. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1051–1062, 2014.

[50] Y. Li, H. Yang, and H. V. Jagadish. Nalix: an interactive natural language interface for querying xml. In *SIGMOD Conference*, pages 900–902, 2005.

[51] Y. Li, H. Yang, and H. V. Jagadish. Nalix: A generic natural language search environment for XML data. *ACM Trans. Database Syst.*, 32(4), 2007.

[52] Y. Li, C. Yu, and H. V. Jagadish. Schema-free xquery. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 72–83, 2004.

[53] Y. Li, C. Yu, and H. V. Jagadish. Enabling schema-free xquery with meaningful query focus. *VLDB J.*, 17(3):355–377, 2008.

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

[55] P. Liang, M. I. Jordan, and D. Klein. Learning dependency-based compositional semantics. In *ACL*, 2011.

[56] T. Liu. *Learning to Rank for Information Retrieval*. Springer, 2011.

[57] Z. Lu, H. Li, and B. Kao. Neural enquirer: learning to query tables in natural language. *IEEE Data Eng. Bull.*, 39(3):63–73, 2016.

[58] A. Markowetz, Y. Yang, and D. Papadias. Keyword search on relational data streams. In *SIGMOD Conference*, pages 605–616, 2007.

[59] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[60] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111–3119, 2013.

[61] M. Minock. A STEP towards realizing codd's vision of rendezvous with the casual user. In *PVLDB*, pages 1358–1361, 2007.

[62] L. Mohan and R. L. Kashyap. A visual query language for graphical interaction with schema-intensive databases. *IEEE Trans. Knowl. Data Eng.*, 5(5):843–858, 1993.

[63] N. Nakashole, G. Weikum, and F. M. Suchanek. Discovering semantic relations from the web and organizing them with PATTY. *SIGMOD Record*, 42(2):29–34, 2013.

[64] A. Nandi and H. V. Jagadish. Qunits: queried units in database search. In *CIDR*, 2009.

[65] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web, 1999.

[66] Y. Papakonstantinou, M. Petropoulos, and V. Vassalos. QURSED: querying and reporting semistructured data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 192–203, 2002.

[67] M. Petropoulos, V. Vassalos, and Y. Papakonstantinou. XML query forms (xqforms): Declarative specification of XML query interfaces. In *Proceedings of the Tenth International World Wide Web Conference, WWW 10, Hong Kong, China, May 1-5, 2001*, pages 642–651, 2001.

[68] A.-M. Popescu, A. Armanasu, O. Etzioni, D. Ko, and A. Yates. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *COLING*, 2004.

[69] A.-M. Popescu, O. Etzioni, and H. A. Kautz. Towards a theory of natural language interfaces to databases. In *IUI*, pages 149–157, 2003.

[70] S. Reddy, O. Täckström, M. Collins, T. Kwiatkowski, D. Das, M. Steedman, and M. Lapata. Transforming dependency structures to logical forms for semantic parsing. *TACL*, 4:127–140, 2016.

[71] T. Report. Template-based natural language interface for relational databases.

[72] S. B. Roy, H. Wang, G. Das, U. Nambiar, and M. K. Mohania. Minimum-effort driven dynamic faceted search in structured databases. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management, CIKM 2008, Napa Valley, California, USA, October 26-30, 2008*, pages 13–22, 2008.

[73] S. B. Roy, H. Wang, U. Nambiar, G. Das, and M. K. Mohania. Dynacet: Building dynamic faceted search systems over databases. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 1463–1466, 2009.

[74] A. Simitsis, G. Koutrika, and Y. E. Ioannidis. Précis: from unstructured keywords as queries to structured databases as answers. *VLDB J.*, 17(1):117–149, 2008.

[75] M. Singh, A. Nandi, and H. V. Jagadish. Skimmer: rapid scrolling of relational query results. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 181–192, 2012.

[76] E. Stoica, M. A. Hearst, and M. Richardson. Automating creation of hierarchical faceted metadata structures. In *Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, April 22-27, 2007, Rochester, New York, USA*, pages 244–251, 2007.

[77] M. Tan, C. N. dos Santos, B. Xiang, and B. Zhou. Improved representation learning for question answer matching. In *ACL*, 2016.

[78] L. R. Tang and R. J. Mooney. Using multiple clause constructors in inductive logic programming for semantic parsing. In *EMCL*, pages 466–477, 2001.

[79] S. Tata and G. M. Lohman. Sqak: doing more with keywords. In *SIGMOD Conference*, pages 889–902, 2008.

[80] Y. W. Wong and R. J. Mooney. Learning synchronous grammars for semantic parsing with lambda calculus. In *ACL*, 2007.

[81] P. Wu, Y. Sismanis, and B. Reinwald. Towards keyword-driven analytical processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 617–628, 2007.

[82] Z. Wu and M. S. Palmer. Verb semantics and lexical selection. In *ACL*, pages 133–138, 1994.

[83] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *ACM Trans. Database Syst.*, 36(3):15, 2011.

[84] D. Xin, Y. He, and V. Ganti. Keyword++: A framework to improve keyword search over entity databases. *PVLDB*, 3(1):711–722, 2010.

[85] Y. Xu, L. Mou, G. Li, Y. Chen, H. Peng, and Z. Jin. Classifying relations via long short term memory networks along shortest dependency paths. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, pages 1785–1794, 2015.

[86] M. Yang, N. Duan, M. Zhou, and H. Rim. Joint relational embeddings for knowledge-based question answering. 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 645–650, 2014.

[87] X. Yao and B. V. Durme. Information extraction over structured data: Question answering with freebase. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, June 22-27, 2014, Baltimore, MD, USA, Volume 1: Long Papers*, pages 956–966, 2014.

[88] K. Yee, K. Swearingen, K. Li, and M. A. Hearst. Faceted metadata for image search and browsing. In *Proceedings of the 2003 Conference on Human Factors in Computing Systems, CHI 2003, Ft. Lauderdale, Florida, USA, April 5-10, 2003*, pages 401–408, 2003.

[89] W. Yih, M. Chang, X. He, and J. Gao. Semantic parsing via staged query graph generation: Question answering with knowledge base. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, pages 1321–1331, 2015.

[90] C. Yu and H. V. Jagadish. Schema summarization. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 319–330, 2006.

[91] C. Yu and H. V. Jagadish. Querying complex structured databases. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1010–1021, 2007.

[92] J. X. Yu, L. Qin, and L. Chang. *Keyword Search in Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[93] J. X. Yu, L. Qin, and L. Chang. Keyword search in relational databases: A survey. *IEEE Data Eng. Bull.*, 33(1):67–78, 2010.

[94] J. M. Zelle and R. J. Mooney. Learning to parse database queries using inductive logic programming. In *AAAI*, 1996.

[95] D. Zeng, K. Liu, Y. Chen, and J. Zhao. Distant supervision for relation extraction via piecewise convolutional neural networks. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, pages 1753–1762, 2015.

[96] D. Zeng, K. Liu, S. Lai, G. Zhou, and J. Zhao. Relation classification via convolutional deep neural network. In *COLING 2014, 25th International Conference on Computational Linguistics, Proceedings of the Conference: Technical Papers, August 23-29, 2014, Dublin, Ireland*, pages 2335–2344, 2014.

[97] Y. Zeng, Z. Bao, T. W. Ling, H. V. Jagadish, and G. Li. Breaking out of the mismatch trap. In *ICDE*, pages 940–951, 2014.

[98] Y. Zhang, K. Liu, S. He, G. Ji, Z. Liu, H. Wu, and J. Zhao. Question answering over knowledge base with neural attention combining global knowledge information. *CoRR*, abs/1606.00979, 2016.

[99] W. Zheng, L. Zou, X. Lian, J. X. Yu, S. Song, and D. Zhao. How to build templates for RDF question/answering: An uncertain graph similarity join approach. In *SIGMOD*, 2015.

[100] M. M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.

[101] M. M. Zloof. Office-by-example: A business language that unifies data and word processing and electronic mail. *IBM Systems Journal*, 21(3):272–304, 1982.