

A Natural Language Interface for Querying Graph Databases

by

Christina Sun

S.B., Computer Science and Engineering
Massachusetts Institute of Technology, 2017

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

© Massachusetts Institute of Technology 2018. All rights reserved.

Author
Department of Electrical Engineering and Computer Science
May 25, 2018

Certified by
Sanjeev Mohindra
Assistant Group Leader, Intelligence and Decision Technologies,
MIT Lincoln Laboratory
Thesis Supervisor

Accepted by
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

A Natural Language Interface for Querying Graph Databases

by

Christina Sun

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2018, in Partial Fulfillment of the
Requirements for the Degree of
Master of Engineering in Computer Science and Engineering

Abstract

An increasing amount of knowledge in the world is stored in graph databases. However, most people have limited or no understanding of database schemas and query languages. Providing a tool that translates natural language queries into structured queries allows people without this technical knowledge or specific domain expertise to retrieve information that was previously inaccessible. Many existing natural language interfaces to databases (NLIDB) propose solutions that may not generalize well to multiple domains and may require excessive feature engineering, manual customization, or large amounts of annotated training data. We present a method for constructing subgraph queries which can represent a graph of activities, events, persons, behaviors, and relations, for search against a graph database containing information from a variety of data sources. Our model interprets complex natural language queries by using a pipeline of named entity recognition and binary relation extraction models to identify key entities and relations corresponding to graph components such as nodes, attributes, and edges. This information is combined in order to create structured graph queries, which may then be applied to graph databases. By breaking down the translation task into a pipeline of several submodules, our model achieves a prediction accuracy of 46.9 % with a small training set of only 218 sentences.

Thesis Supervisor: Sanjeev Mohindra

Title: Assistant Group Leader, Intelligence and Decision Technologies,
MIT Lincoln Laboratory

Acknowledgments

I would like to thank Dr. Sanjeev Mohindra for supervising my thesis and providing a project that allowed me to explore a variety of new research topics from which I learned a great deal. I would also like to express my gratitude to Dr. Michael Yee and Andrew Heier for their encouragement and guidance in all aspects of this thesis, from narrowing down a project topic, to implementing possible solutions, and everything in between. Special thanks to the MIT Supercloud team for providing computing resources and the developers of MIT Information Extraction (MITIE) for providing state of the art information extraction tools. Finally, I would not have completed this thesis without the unconditional support of my parents, friends and family.

Contents

1	Introduction	13
1.1	Motivations of this work	13
1.2	Motivation for using natural language processing in querying graph databases	14
1.3	Thesis outline	17
2	Review of Existing Work	19
2.1	Review of natural language processing strategies in querying databases . . .	19
2.1.1	Pattern matching systems	20
2.1.2	Syntax-based systems	20
2.1.3	Semantic grammar systems	21
2.1.4	Intermediate representation languages	22
2.1.5	Sequence to sequence models	23
2.2	Review of graph databases and related modules	24
2.2.1	Comparison of graph databases and relational databases	24
2.2.2	Subgraph matching	25
2.2.3	Existing graph query languages	25
2.2.4	Complexities of existing graph query languages	26
2.2.5	Visual query builders	27
3	Model Architecture	29
3.1	Dataset	31
3.2	Model pipeline overview	34
3.3	Information extraction phase	34

3.3.1	Training data	34
3.3.2	Named entity recognition	36
3.3.3	Binary relation extraction	38
3.3.4	Output re-organization	40
3.4	Information processing phase	40
3.4.1	Node processing	40
3.4.2	Attribute processing	41
3.4.3	Edge processing	42
3.5	Post processing phase	44
3.5.1	User feedback and cleanup	44
4	Results	47
4.1	Dataset	47
4.2	Information extraction phase evaluation	48
4.3	Information processing phase evaluation	53
4.4	Qualitative analysis	56
4.4.1	Word embeddings	57
4.4.2	Multiple attributes	57
4.4.3	Implicit nodes and edges	57
5	Conclusion and Future Work	61
5.1	Conclusion	61
5.2	Future work	62
A	Pseudocode for model pipeline	65

List of Figures

1-1	Example of a graph containing information from multiple data sources. . . .	14
1-2	The query translation engine within the context of a broader graph querying system.	16
2-1	A sentence represented as a parse tree in a syntax-based system (left) and in a semantic grammar system (right).	21
3-1	An example of a graph database.	33
3-2	The model pipeline. Information extracted in the first phase is color coded to depict where it is used as input to the second phase. The shaded boxes represent sub modules within the pipeline. The non-shaded boxes represent the inputs and outputs at each step.	35
4-1	Average precision for the NER models using various amounts of training data.	50
4-2	Average precision for the BRE models using various amounts of training data.	50
4-3	Average recall for the NER models using various amounts of training data. .	51
4-4	Average recall for the BRE models using various amounts of training data. .	51
4-5	Average F1 score for the NER models using various amounts of training data.	52
4-6	Average F1 score for the BRE models using various amounts of training data.	52

List of Tables

1.1	A natural language query and its corresponding structured language translation and graphical representation.	16
2.1	Examples of queries represented in Gremlin [7], Cypher [20], and SPARQL [24].	27
3.1	Node types, edge types, and attributes of the graph.	33
3.2	The input and outputs of the named entity recognition models. The input is the natural language query. The outputs are entities, their positions in the sentence, and their tags.	37
3.3	The inputs and outputs of the binary relation extraction models. The inputs are the trained NER-G model and a set of entity pairs. The output of each binary relation extractor is a set of scores associated with each combination of ordered endpoint nodes.	39
4.1	The amount of training data available for each named entity recognition (NER) module and each binary relation extraction (BRE) module.	48
4.2	K-fold cross validation scores for each model (k=5).	49
4.3	The effect of each model in the information extraction phase on the structured language prediction scores (precision, recall, and F1 for nodes and edges, overall query accuracy and edit cost).	54
4.4	The effect of each model in the information extraction phase on the structured language prediction scores (precision, recall, and F1 for nodes and edges, overall query accuracy and edit cost).	55

4.5 The output of each submodule in the pipeline and the final structured language prediction for an example natural language input query. 58

Chapter 1

Introduction

1.1 Motivations of this work

Large quantities of information are collected and stored in databases every day. With rapid growth in fields ranging from artificial intelligence to the Internet of Things, this data has the potential to advance our knowledge further, especially if it is accessible for people to use and analyze. Unstructured data is constantly collected in various formats such as text, images, audio, and video, from a multitude of sources including social media, online encyclopedias, and more. Structured data containing information about entities and their relationships are often gathered automatically from unstructured text or images, for example via photo tagging in social media websites, or manually, as in the case of Wikipedia information boxes.

Graph databases are useful for representing, organizing and analyzing data, and for fusing different datasets together into a common format. For example, analysts may aggregate information gathered from Twitter, Facebook, and LinkedIn into a single graph for community detection and other types of social network analysis. Other applications include using product and customer data in a graph database for recommendation engines, analyzing anomalies among relationships for fraud detection, and mapping communications networks for outage prediction. An example of a graph containing information pulled from different sources is shown in Figure 1-1. This example shows that “ANDREW NG” and “MICHAEL JORDAN” are co-authors of the paper “LATENT DIRICHLET ALLOCATION,”

and that “MICHAEL JORDAN” and “ANTHONY JOSEPH” both work at “UC BERKELEY.” The “CO-AUTHOR” and “WROTE” relations are extracted from a database of computer science journals and proceedings, and the employer-employee relations are extracted from a source containing information about professional networks.

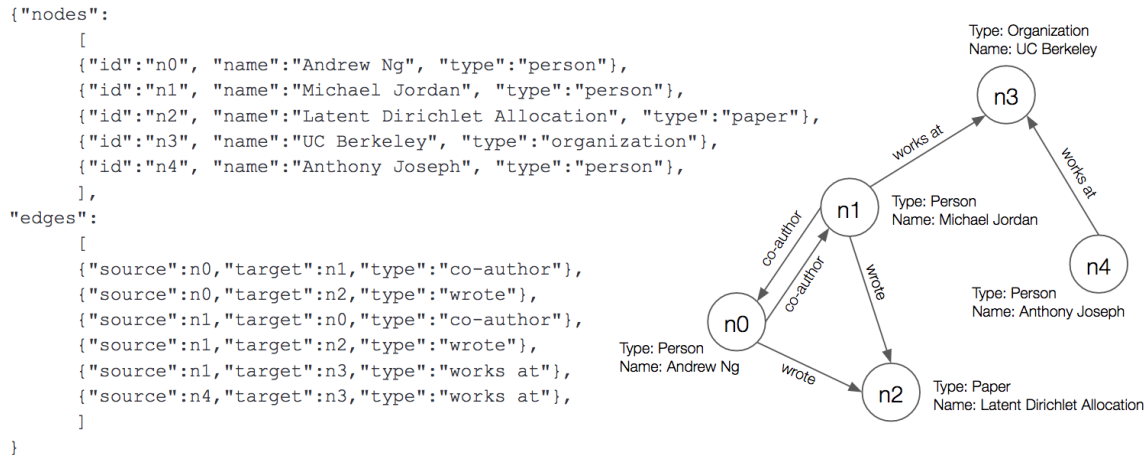


Figure 1-1: Example of a graph containing information from multiple data sources.

Graph databases demonstrate relationships among entities in an intuitive way that often offers more flexibility and higher performance compared to other types of databases. They denote explicit relationships and directional activity, which are difficult to encode at scale in relational databases. However, challenges in working with graph databases may arise due to the user’s unfamiliarity with database schemas, and the complexity and multitude of available query languages. Our primary motivation is to allow analysts who may not necessarily have technical knowledge of graph databases or specific domain expertise to tap into this wealth of data, shifting their time from information retrieval to information analysis.

1.2 Motivation for using natural language processing in querying graph databases

Natural language interface to database systems produce database queries by translating natural language sentences into a structured format which in our case, is a subgraph query.

They play an increasingly important role as people, not only those with specific domain expertise or knowledge of structured query languages, seek to obtain information from databases. These databases contain massive amounts of data and are expanding rapidly, especially since text and voice interfaces have exploded in popularity due to the accessibility and prevalence of web and mobile technologies.

A tool that offers quick access to information in a graph containing data possibly from multiple sources would benefit analysts from various backgrounds, allowing them to focus their resources on other tasks. Many current systems require that the user specify relationships between edges and nodes in order to query the graph. A natural language based system must therefore correctly interpret the sentence, remove extraneous information, map tokens and phrases to nodes and edges, and resolve any ambiguities.

We approach this problem by implementing a pipeline of traditional natural language processing and machine learning components for prediction and classification tasks. Our solution requires a small training set, in contrast to neural models which are expensive in terms of time and cost to train. Given a source query expressed in natural language, our goal is to produce a target query expressed in structured language (a subgraph query in our case). In the example shown in Table 1.1, the natural language sentence, “SHOW ME ALL ARTICLES IN NIPS WRITTEN BY SOMEONE FROM GOOGLE BRAIN” is transformed into a subgraph query with nodes representing the article, author, venue, and organization, and edges representing the relationships between nodes. The target query describes the subgraph components in a structured format. This task is an example of structured prediction, which refers to the prediction of structured objects rather than the prediction of discrete or continuous values. However, we decompose the prediction process into several sub-components instead of computing the output in one shot. This allows each sub-component to focus on one part of the broader prediction problem, resulting in higher overall accuracy.

Existing systems have used pattern matching, syntax-based, and semantic grammar methods to produce intermediate query representations. More recently, advances in natural language processing and machine learning tasks have offered new directions for improvement. For example, sequence to sequence methods [10, 3, 30, 16] have enjoyed success in neural machine translation. We discuss the advantages and drawbacks of these methods in

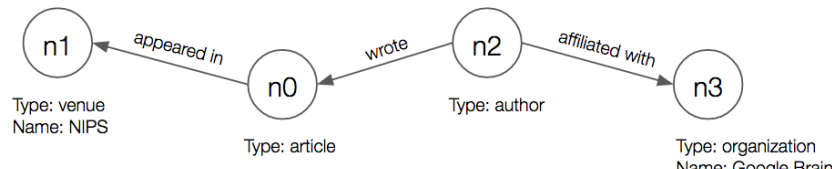
Source	“Show me all articles in NIPS written by someone from Google Brain.”
Target	<pre> `nodes` : [{`attributes`:[], `type`:`paper`, `id`:`n0`}. {`attributes`:[{`name`:`venue_name`, `value`:`NIPS`, `op`:`=`}], `type`:`venue`, `id`:`n1`}, {`attributes`:[], `type`:`author`, `id`:`n2`}, {`attributes`:[{`name`:`organization_name`, `value`:`Google Brain`, `op`:`=`}], `type`:`organization`, `id`:`n3`},], `edges` : [`to`:`n1`, `from`:`n0`, `type`:`appeared`, `to`:`n0`, `from`:`n2`, `type`:`wrote`, `to`:`n3`, `from`:`n2`, `type`:`affiliated`,], </pre>
Subgraph	

Table 1.1: A natural language query and its corresponding structured language translation and graphical representation.

greater detail in the next chapter. The scope of this work is the development of the query translation engine shown in Figure 1-2.

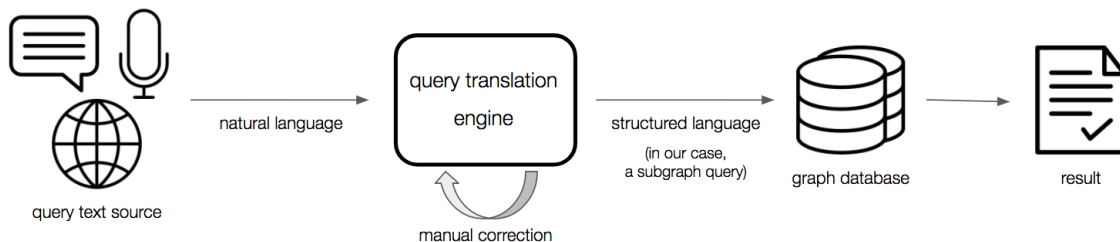


Figure 1-2: The query translation engine within the context of a broader graph querying system.

1.3 Thesis outline

This thesis presents a natural language interface for querying graph databases. Chapter One describes the motivation and gives an introduction for using natural language processing techniques in the task of querying graph databases. We review sub-tasks and existing systems related to solving the associated issues in the second chapter. The third chapter provides a description of each module in our proposed pipeline as well as the natural language processing algorithms involved. Chapter Four discusses the results and evaluation of each module as well as the system as a whole. In the final chapter, we draw conclusions and propose topics for future work.

Chapter 2

Review of Existing Work

2.1 Review of natural language processing strategies in querying databases

Natural language interface to database systems have evolved over decades of research. Most existing research in this field has been conducted within the context of querying relational databases, which are widely used today. Due to the success and prevalence of relational databases, SQL has emerged as a relatively standardized query language. In comparison, the popularity of graph databases has risen more recently, and a standardized system akin to SQL for relational databases has not yet emerged.

Natural language interfaces for graph databases face challenges related to other natural language processing tasks. For example, in Question Answering (QA), the system must also correctly interpret a natural language input question in order to produce the desired result. This field is closely related to human-computer interaction, in addition to natural language processing and relational databases. Androutsopoulos et al. [2] review some methods typically used in this problem space, which we will briefly summarize along with a description of more recent techniques.

2.1.1 Pattern matching systems

Pattern matching systems look for exact keyword matches in a sequence of tokens. As an example, one could engineer a rule such that “*capital of Italy*” and any variant of a natural language query containing the word “*capital*” followed by a country name would be mapped to a structured query of the form, `Report Capital of row where Country = <Italy>`. The ELIZA [28] chatbot demonstrated the ability of a natural language interface to successfully simulate conversation with a human user by using pattern matching and substitution methods.

Pattern matching approaches benefit from simplicity and ease of implementation. However, they face difficulties when processing complex queries. In many cases, sentence ambiguities are easily resolved by understanding the context or underlying sentence structure. Androutsopoulos et al. [2] demonstrate how a pattern matching system processing a query such as “*CITIES IN SWEDEN*” may confuse the word “*IN*” with the location “*Indiana*.” Our approach avoids this issue by making a first attempt at resolving entities and relations, and would tag “*IN*” as a location relation. In a sentence where the context makes it clear that “*IN*” refers to a location, the system would tag the token as a location entity. Therefore, our system utilizes more sophisticated techniques to better handle sentence ambiguities.

2.1.2 Syntax-based systems

Syntax-based systems use a specified grammar to parse the user’s natural language query and represent it as a parse tree. This parse tree is then mapped to an expression in the desired structured database query language. Syntax-based approaches identify detailed information about the underlying sentence structure such as the part of speech of each word, the phrase type of groups of words, and ways to group phrases into even more complex structures which form the sentence.

LUNAR [29] is an example of one such natural language interface which allowed researchers to ask questions in the domain of lunar geology. Syntax-based interfaces such as LUNAR are often domain specific and require extensive manual engineering in order to create mapping rules that transform parse trees into structured language queries. Since

our dataset includes data from multiple sources, we seek a solution that can generalize to multiple domains. Another disadvantage of such systems is that multiple parse trees may correspond to the same sentence. Even if the system is able to map the parse tree to a structured language query, it could produce multiple valid queries, and it is difficult to determine which is correct without additional information.

2.1.3 Semantic grammar systems

Semantic parsing is a related task commonly used in natural language processing problems such as question answering. Natural language questions are broken into predetermined semantic categories and parsed into formal representations. In question answering, these representations are then executed on a database. Semantic parsing methods are often restrictive in that they operate on a single schema, and require a person to manually engineer grammars. LADDER [8] and PLANES [17] are examples of natural language interface to database systems that use semantic grammar concepts.

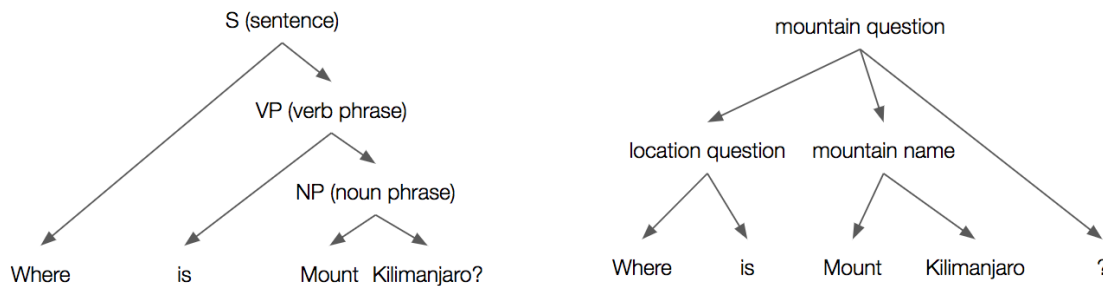


Figure 2-1: A sentence represented as a parse tree in a syntax-based system (left) and in a semantic grammar system (right).

As shown in Figure 2-1, semantic grammar systems are similar to syntax-based systems in that they transform a natural language query into a parse tree. In this example, the sentence to be parsed is, “*Where is Mount Kilimanjaro?*” The syntax-based system categorizes “*Mount Kilimanjaro*” as a noun phrase. It then combines “*is*” with the noun phrase to form a verb phrase. Finally, “*Where*” and the verb phrase form a sentence. The semantic grammar system categorizes “*Where is*” as a location question and “*Mount Kilimanjaro*” as a mountain name. The combination of location question and mountain name categories

form a mountain question. In a semantic grammar system, the categories do not necessarily correspond to syntactic concepts such as noun phrases and verb phrases. Instead, the parser uses semantic concepts specific to the question domain. While syntactic concepts are relevant for all sentences, semantic grammar categories are richer in meaning, eliminating the less useful nodes in syntax-based systems. Furthermore, semantic grammars allow for more flexibility and ease in mapping the parse tree to a structured language representation because the categories are defined for the use case, rather than by the English language.

One weakness of semantic grammar methods is that generating the mappings to structured language require knowledge of entities in the domain. Furthermore, the broader semantic categories used in one domain most likely generalize poorly to other domains. Our approach generalizes to different schema and allows someone without extensive information about the dataset to be able to retrieve data.

2.1.4 Intermediate representation languages

Many intermediate representation languages address the difficulties found in previous approaches of translating parse trees into structured database query languages by instead breaking the problem into simpler steps. Some models may chain together several intermediate representations in order to transform a natural language query into its corresponding structured language query. For example, the user's question may be parsed syntactically, resulting in a parse tree which is then translated using a semantic interpreter into an intermediate representation such as a logical query representation. This intermediate representation is then mapped to the final structured query format.

Examples of interfaces that use intermediate representation languages are Edite [25] and System X [4]. Androutsopoulos et al. [1] use an intermediate representation language approach in which the sentence is first transformed into a logical query. The resulting logical query is then transformed into a database query. The intermediate logical query expresses the natural language query in terms that are common to all database structures, yet are still meaningful. This balances the trade-offs between syntax-based systems and semantic-based systems, where sentences are broken into chunks that are either universal

to all sentences but not meaningful enough to easily map to a structured language format, or rich in meaning but not generalizable to sentences in other domains. Another advantage of using logic query languages is that they allow for reasoning capabilities. Logic query languages do not depend on the database used, and can therefore work in different domains and with different query languages.

2.1.5 Sequence to sequence models

More recently, researchers have successfully applied sequence to sequence models to tasks such as machine translation. As a result, these models are gaining popularity in other natural language processing tasks as well. Sequence to sequence models convert a sequence in one domain into a sequence in a different domain by passing the input sequence through an encoder to obtain hidden states, and using a decoder to produce the output sequence from the hidden states. The task of translating natural language queries into structured language queries can be viewed as an application of neural machine translation.

Brad et al. [3] use OpenNMT [16], an open source sequence to sequence machine translation model implementation with attention, to translate natural language queries into SQL queries. Since these models do not involve any intermediate representations and instead are more end to end, they can be rapidly deployed for a variety of target domains. However, they require a large training set of 24,890 natural language queries and their SQL translations, which is costly and time-consuming to obtain. Iyer et al. [10] extend the use of deep neural sequence models by incorporating user feedback to flag incorrect queries. These annotations are readily available due to the popularity of SQL. The authors also use entity anonymization and provide templates for initially bootstrapping the model.

Zhong et al. [30] propose a sequence to sequence model that translates natural language queries to SQL queries. Since the target language is structured, the authors restrict the output space of the neural model. This results in queries that are much more likely to be correct, and thus directly executable. They also implement a pointer network which generates the output sequence by pointing to tokens in the input sequence, a concatenation of the column names, the condition columns of the query, the question, and the target

language vocabulary. This approach may not work well with previously unseen terms, since the output is derived from a predefined vocabulary. Again, this method requires a large training set. The authors use a dataset consisting of 80,654 pairs.

Most current sequence to sequence approaches such as those described above translate natural language queries into structured queries that are applied to traditional relational databases. Relational databases operate efficiently when the data is easily mapped into tables, columns, and rows, and when queries do not require that the system joins together many separate tables. In our problem setting, users query graph databases composed of information from possibly many different domains. While other strategies restrict the query domain to single tables, our approach must handle graphs built from several data sources.

2.2 Review of graph databases and related modules

2.2.1 Comparison of graph databases and relational databases

Most of the approaches described in the previous section focus on querying relational databases. Our problem setting focuses on the related problem of querying graph databases. We describe the main differences between the two.

A relational database stores data in tables where each row represents a record and each column corresponds to an attribute of a record. Therefore, a table, or relation, represents a set of records which share the same attributes. SQL is used to query and process data in most relational database systems. While relational databases are highly structured and benefit from having a nearly universal query language, there are some drawbacks to using them. In order to analyze relationships among entities across different tables, the expensive “join” operation is used to combine relations. This operation is expensive because it requires index lookups and matching in order to find a related column in the tables.

Graph databases prioritize modeling relationships among entities. They store entities and their relationships as nodes and edges, which may be augmented with various attributes. Therefore, retrieving the edge between two entities does not involve the expensive “join” operation that is necessary when the data is stored in a relational database. Instead,

the system simply traverses the graph, since edges are stored along with nodes when they are inserted. In other words, processing data in a graph benefits from “index-free adjacency,” the property that each node is stored with its adjacent nodes and edges. This results in efficient relationship retrieval even for complex queries, as the number and depth of relationships increase. Graph databases also model data in a visually intuitive manner. They are appropriate in our problem setting in which users draw insights from multiple data sources fused into a common graph, since relationships between entities are of high importance. Examples of graph databases used today include Neo4j, Titan, and Blazegraph.

2.2.2 Subgraph matching

Exact subgraph matching or subgraph isomorphism search tackles the problem of finding all possible instances of a query graph within a larger graph. In the broader graph querying system, this step uses the structured output of the query translation engine to obtain a result from the graph database. Many current solutions filter intermediate results in order to reduce the search space and processing times. Nabti et al. [19] develop a method that reduces computational load. However, we aim to effectively and accurately search the space by directly asking the user for input to narrow down the options. Our system aids the user in formulating an accurate structured query, which is then executed against the graph database. It is often impossible to find an exact subgraph match. In these cases, inexact subgraph matching algorithms such as G-Ray [27] may still return a result by relaxing the requirement of an exact match and instead retrieve close matches.

2.2.3 Existing graph query languages

Gremlin

Gremlin [26] is the graph traversal language for Apache Tinkerpop, an open source graph computing framework similar to Oracle’s JDBC for SQL databases. It allows users to traverse complex property graphs by constructing queries composed of a sequence of steps. Each step either transforms the objects in, removes objects from, or computes statistics about the data stream.

SPARQL

SPARQL is a query language for systems that contain database information stored in the Resource Description Framework (RDF) format. RDF is a graph-based data format for modeling information related to web resources. Users specify `SELECT`, `CONSTRUCT`, `ASK`, and `DESCRIBE` queries to extract raw values, transform information into RDF, obtain True/False results for a query, and more.

Cypher

Cypher is a query language inspired by SQL, for the widely used Neo4j graph database management system. Cypher describes graph patterns in a visual format reminiscent of ASCII art syntax. It allows users to perform operations such as selecting, inserting, deleting, or updating data in the graph.

2.2.4 Complexities of existing graph query languages

Table 2.1 presents examples of existing graph query commands in Gremlin, Cypher, and SPARQL. The Gremlin query searches for people (managers in the management chain from "gremlin" to the CEO), the Cypher query searches for a relation that satisfies a certain property, and the SPARQL query searches for a book. As shown in the table, each language uses a different approach for querying the database. Gremlin systematically traverses the graph, Cypher visually depicts edges between nodes with arrows, and SPARQL uses syntax reminiscent of SQL. Casual users may find it difficult to produce such queries without first referring to documentation. In addition, there may be multiple ways to construct the same query. Our system abstracts away the details of graph query languages, allowing the user to directly query the system in natural language without worrying about learning the syntax of unfamiliar query languages. The output produced is an intermediate format that can easily be mapped to formal query languages.

Gremlin	<pre>g.V().has("name", "gremlin"). repeat(in("manages")). until(has("title", "ceo")). path().by("name")</pre>
Cypher	<pre>MATCH (n1:Label1)-[rel:TYPE]->(n2:Label2) WHERE rel.property > {value} RETURN rel.property, type(rel)</pre>
SPARQL	<pre>PREFIX dc: <http://purl.org/dc/elements/1.1/> PREFIX : <http://example.org/book/> SELECT \$title WHERE { :book1 dc:title \$title }</pre>

Table 2.1: Examples of queries represented in Gremlin [7], Cypher [20], and SPARQL [24].

2.2.5 Visual query builders

Aids such as visual query builders are often used to assist users with designing and constructing valid queries. The prevalence of visual query builders extends to several query languages. Existing systems for querying databases via visual query builders [11, 5, 12, 13, 9] involve sketching nodes and edges which are subsequently translated into structured language queries.

Visual query builders can be tedious to use, especially if the user is unfamiliar with the database schema. Learning to use the tools themselves often requires specific technical knowledge, such as what is classified as a node, attribute, or edge. In the case where the user has limited knowledge of the schema and possible values, it may be more convenient to directly write the query rather than deal with a visual query builder. These issues further motivate natural language interfaces to databases. A natural language interface requires no technical knowledge from the user about how to express a query.

Chapter 3

Model Architecture

Our proposed model translates natural language queries into structured language queries for graph databases. An example of a natural language query and its translation is shown in Table 1.1. At a high level, the system must extract tokens from the source sentence corresponding to the nodes, attributes, and edges, and differentiate between the three. The system must then determine how to build the overall subgraph structure from the extracted information. There is no guarantee that the user is familiar with the data and the schema. Therefore, the system first attempts to build a valid query with the user's input. The user may then provide feedback, resolving any ambiguities and errors that may arise due to limitations in the system's ability to interpret variations in the way each user expresses a query in natural language, or other challenges. For example, if the system produces the result [{"name": "organization_name", "value": "ICML", "op": "="}], but ICML refers to a venue rather an organization, the user can manually correct the query to [{"name": "venue_name", "value": "icml", "op": "="}].

Challenges

Although the natural language form of a query is often understandable and unambiguous in that the meaning is well-defined regardless of data modeling choices, an automated system

may still find it difficult to generate the structured query representation. We discuss some challenges that our system faces.

There are numerous valid ways to express the same query in natural language, and some are easier for the system to process and map to structured language than others. For most natural language queries, the desired answer is usually clear and unambiguous to a human user despite variation in the way the query is expressed. However, it is difficult to judge what types of sentences the system will struggle with, especially without knowledge of how the model works. For example, the queries “PAPERS WRITTEN BY SMITH AND ALLEN” and “PAPERS WRITTEN BY SMITH AND WRITTEN BY ALLEN” are essentially equivalent to a human user, but simple systems may face greater difficulty in interpreting one over the other. Over time, users may learn the types of sentences that the system handles correctly, similar to how smart phone users adapt to using voice commands that most often return the desired behavior. Still, the system should be robust to a diverse user base.

The system can fail to return the correct result for a variety of reasons, and users usually have no method for diagnosing the issue. Natural language interfaces that behave as black boxes prevent the user from knowing if failure occurred because the desired result is not in the database, or because there are linguistic issues in the query, in which case properly rephrasing the query could result in the correct answer.

Previous natural language interface to database systems faced a trade-off between domain independence and performance. While engineering specific components of the system with domain knowledge in mind may improve the system’s ability to correctly interpret a query and return a valid result, this strategy fails when the user must query data from multiple domains.

Linguistic issues include coreference, anaphora, word-sense disambiguation, and more. Coreference refers to multiple tokens in the sequence referring to the same entity. In the sentence “MT. EVEREST IS LOCATED IN THE HIMALAYAS AND IT IS THE HIGHEST MOUNTAIN ON EARTH,” the tokens “MT. EVEREST” and “IT” refer to the same entity.

Anaphora is the use of an expression to implicitly refer to another expression in the sentence. For example, in the sentence “IF MATT BUYS A NEW CAR, I WILL DO IT AS WELL,” the expression “DO IT” refers to the action of purchasing a new car. Word-sense disambiguation refers to inferring from context the correct meaning of words that may have multiple definitions. For example, the word “CRANE” can refer to a type of bird or a type of machinery.

Our system handles these issues by showing the user its interpretation of the query and allowing the user to correct any mistakes. This would give insight to the user as to where and why errors may arise. We handle linguistic problems through a pipeline of state of the art natural language processing modules designed to deal with each, including several named entity recognizers and binary relation extractors.

3.1 Dataset

Several public datasets are available for use in developing natural language interfaces for querying databases. The Stack Exchange Natural Language Interface to Database (SENLIDB) corpus [3] includes 24,890 (natural language text, SQL query) pairs constructed using the Stack Exchange API. WikiSQL [30] contains 80,654 pairs of the same form derived from 24,241 Wikipedia tables. These datasets are appropriate for the task of translating natural language queries into their corresponding structured language queries. However, these datasets are relevant for constructing SQL queries and querying relational databases, rather than for querying graph databases.

There are a number of other datasets which contain data that is relevant to the tasks that users will eventually use our model for. For example, the NYC Taxi and Limousine Commission Trip Record Data contains taxi trip data such as pick-up and drop-off dates, times, and locations, and trip distances, fares, rates, payment types, and passenger counts reported by the driver. DBLP contains bibliographic records such as ISBN, title, journal,

author, year, volume, and pages about computer science journals and proceedings. GDELT contains billions of references about people, locations, organizations, events, and more, derived from broadcast, print, and web news sources around the world. A user could query a graph created from fusing these related datasets to answer questions about conference attendees who may have taken taxi rides while a major event occurred in the same city, for example.

We have constructed a property graph dataset that reflects the structure, richness, and diversity of datasets studied by MIT Lincoln Laboratory. A property graph is composed of nodes, which represent objects or entities in the graph, and edges, which represent connections between two nodes. Each node is associated with a unique identifier and a set of attributes, which are represented as key-value pairs. Each edge is associated with a source node, a target node, and a label denoting the relationship between its two endpoints.

The graph, consisting of data from DBLP, is shown in Figure 3-1. Each node is associated with a set of attributes, and the source and target nodes of each edge type are restricted to specific node types. This graph can be expanded to include information from multiple data sources. The system only requires that the node types, edge types, and attribute types are specified beforehand. For our example graph, these specifications are shown in Table 3.1. The node types are **“author,” “paper,” “venue,”** and **“organization,”** and the edge types are **“wrote,” “referenced,” “appeared in,”** and **“affiliated with.”** The attribute types associated with **“author”** nodes are name and ID (a unique identifier), the attribute types associated with **“paper”** nodes are title, year, and ID, the attribute types associated with **“venue”** nodes are name, date, and location, and the attribute types associated with **“organization”** nodes are name and location. The **“wrote”** relation points from an **“author”** node to a **“paper”** node, the **“referenced”** relation points from a **“paper”** node to a **“paper”** node, the **“appeared in”** relation points from a **“paper”** node to a **“venue”** node, and the **“affiliated with”** relation points from an **“author”** node to an **“organization”** node. When a user queries the database, for example searching for a paper with

a certain title or a particular author, the system will retrieve the paper by translating the natural language query into a subgraph query which is then applied to the graph database.

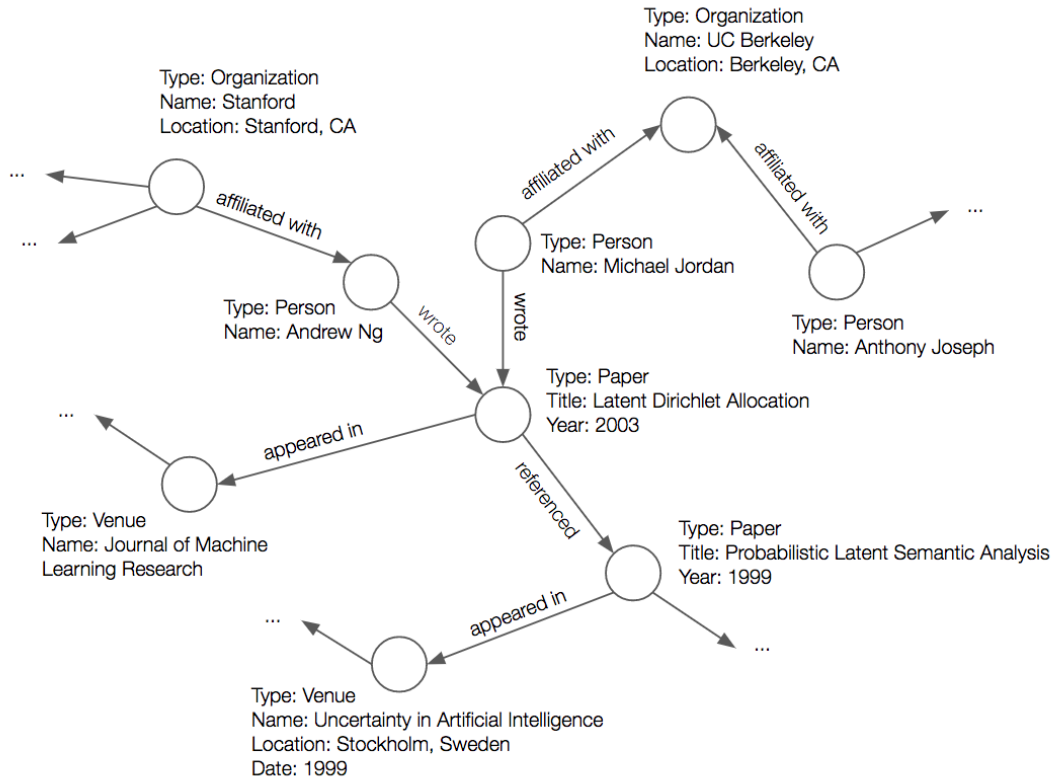


Figure 3-1: An example of a graph database.

Nodes		Edges		
type	attributes	type	source	target
author	name, ID	wrote	author	paper
paper	title, year, ID	referenced	paper	paper
venue	name, date, location	appeared in	paper	venue
organization	name, location	affiliated with	author	organization

Table 3.1: Node types, edge types, and attributes of the graph.

3.2 Model pipeline overview

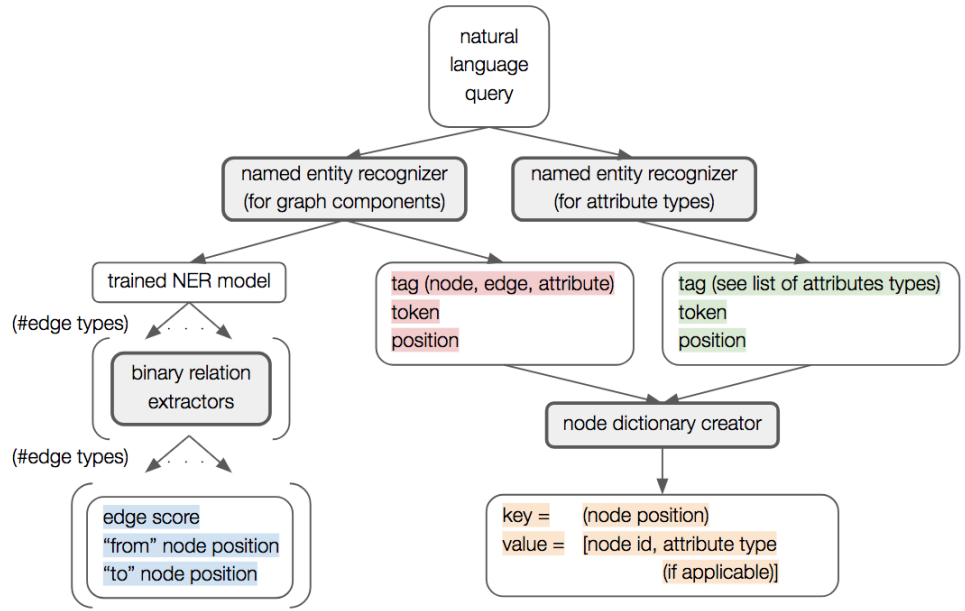
We decompose the process of translating a natural language query into a structured language query into several building blocks. An overview of the model pipeline is shown in Figure 3-2. Figure 3-2a describes the first phase of the pipeline. The purpose of this phase is to use state of the art natural language processing techniques to extract information such as key entities and their positions in the sentence, as well as possible relationships between them. Figure 3-2b describes the second phase of the pipeline, in which we use the data collected in the first phase as input to create the structured graph components (nodes, attributes, and edges) of the query.

Before explaining the sub-modules in detail, we give an overview of how the system processes a natural language query. First, we train two named entity recognizers and n binary relation extractors, where n is the number of edge types in the graph. The first named entity recognizer (NER-G) tags tokens in the sentence that are likely to be graph components, such as nodes, attributes, or edges. The second named entity recognizer (NER-A) tags attribute tokens in the sentence with their corresponding attribute type. Each binary relation extractor produces a score for every combination of two nodes in the sentence. Given an edge type, a higher score for the node combination (NODE A, NODE B) denotes that the likelihood of an edge of that type pointing from NODE A to NODE B is higher. After collecting this information, we look at each node, attribute, and edge detected, and process it accordingly.

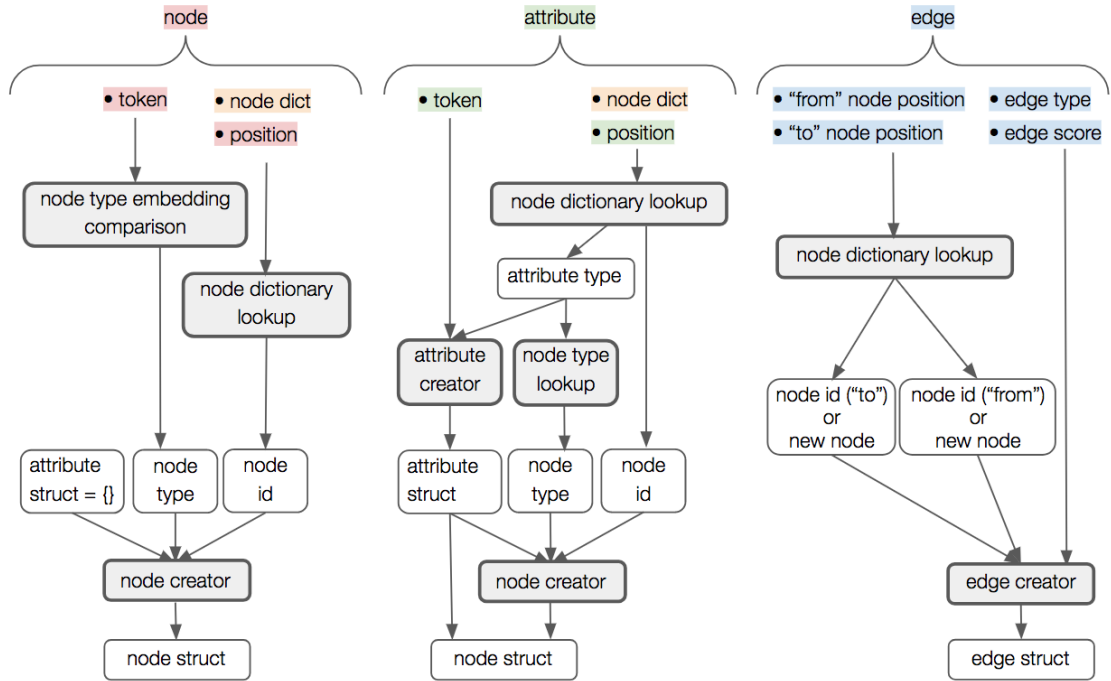
3.3 Information extraction phase

3.3.1 Training data

In order to train the named entity recognition and binary relation extraction modules, we create a small training set of natural language queries. The system should work well with



(a) The information extraction phase of the model pipeline.



(b) The information processing phase of the model pipeline.

Figure 3-2: The model pipeline. Information extracted in the first phase is color coded to depict where it is used as input to the second phase. The shaded boxes represent sub modules within the pipeline. The non-shaded boxes represent the inputs and outputs at each step.

limited amounts of training data, and the training set can be augmented as users enter more sentences into the system over time. The detailed counts are shown in Table 4.1.

Named entity recognition training data

The training data for the named entity recognition model for graph components (NER-G) consists of 218 manually annotated sentences. The average sentence length is 10.07 tokens. Annotating the training sentences results in 1035 training samples. Each sample is an entity which is marked as a node, attribute, or edge. The training data for the named entity recognition model for attribute types (NER-A) consists of the same 218 manually annotated sentences. Annotating the training sentences results in 414 training samples. Each sample is an entity which is marked as one of the attribute types.

Binary relation extraction training data

The training data for each binary relation extraction module is derived from the same 218 manually annotated sentences used for training the named entity recognition modules. Since all edge types may not be present in every sentence, the number of training sentences for each binary relation extractor varies. For each sentence, we compile a list of every token that was marked as a node or an attribute, and produce every combination of two tokens from this list. For each edge type, we annotated on average 804 such pairs, marking each as positive if the pair represented a valid relation, and negative otherwise. The exact counts vary because in natural language, some relations appear more frequently than others.

3.3.2 Named entity recognition

Named entity recognition (NER) is a task commonly used in information extraction for classifying entities of interest into predefined categories, such as the names of people, organizations, or locations. We train two named entity recognition models, whose inputs are

the natural language query. The named entity recognizer for graph components (NER-G) tags tokens in the input sentence as N (node), E (edge), or A (attribute). The named entity recognizer for attribute types (NER-A) tags tokens in the input sentence according to their attribute type (listed in Table 3.1).

Input	“SHOW ME ALL PAPERS CO-AUTHORED BY VINYALS APPEARING IN NIPS.”
NER-G output	N; papers; xrange(3, 4) E; co-authored by; xrange(4, 6) A; Vinyals; xrange(6, 7) E; appearing in; xrange(7, 9) A; NIPS; xrange(9, 10)
NER-A output	author_name; Vinyals; xrange(6, 7) venue_name; NIPS; xrange(9, 10)

Table 3.2: The input and outputs of the named entity recognition models. The input is the natural language query. The outputs are entities, their positions in the sentence, and their tags.

Table 3.2 presents the output of the named entity recognition modules given an example input sentence. The output of the NER-G module is a set of tokens and their corresponding N (node), E (edge), or A (attribute) tags and positions in the sentence. The output of the NER-A module is a set of tokens and their corresponding attribute types and positions in the sentence. For example, the NER-G module tags PAPERS as a node, CO-AUTHORED BY as an edge, VINYALS as an attribute, and so on. The NER-A module tags VINYALS as an author name and NIPS as a venue name.

Named entity recognition model

We use the state of the art named entity recognizer from the open-source MIT Information Extraction (MITIE) toolkit to train both NER models. MITIE was developed by MIT Lincoln Laboratory on top of the high-performance C++ machine learning toolkit Dlib [15], and can be used with C, C++, Java, R, MATLAB, and Python. MITIE’s NER model uses distributional word embeddings [6], Conditional Random Fields, and Structural Support Vector Machines [14] in its implementation. MITIE also offers pretrained models for

English and Spanish trained using various corpora (including Wikipedia, Freebase, and Gigaword).

MITIE uses a chunker to split the tokenized sentence into entities and non-entities using structural support vector machines. A multiclass classifier categorizes each entity into one of $m + 1$ classes, where m is the number of labels provided by the user. The model learns to classify each entity into one of the m classes defined by the user or the one additional class denoting that the entity should not be labeled as one of the m classes. In our example property graph, the NER-G module categorizes entities into 4 classes, and the NER-A module categorizes entities into 11 classes.

The classifier uses features such as whether or not the token is capitalized, contains only capitalized letters, contains numbers, contains letters, contains numbers and letters, contains only numbers, contains hyphens, contains alternating capital letters in the middle, and more. The model uses BOBYQA [23] for automatic tuning of hyperparameters.

3.3.3 Binary relation extraction

Binary relation extraction (BRE) refers to the task of identifying the relationship between two entities. We train one binary relation extractor module for each edge type. The input to each, is the trained NER-G model and a pair of tokens representing nodes. We first compile a list of tokens that were marked as either N (node) or A (attribute). We include tokens that were marked as A in addition to those marked as N because these tokens often stand in for a node in natural language. For example, people often say “JOHN WROTE A REPORT” instead of “THE AUTHOR JOHN WROTE A REPORT”. The system marks JOHN as A, and will create a node with the appropriate attribute in later steps of the pipeline. For every combination of two tokens (n_1, n_2) from this list, each binary relation extractor produces a score which represents how likely it is that an edge of that type points from n_1 to n_2 .

Continuing with our example sentence “SHOW ME ALL PAPERS CO-AUTHORED BY VINYALS APPEARING IN NIPS,” the output of the binary relation extraction modules are

shown in Table 3.3. Since our example graph contains four edge types, we pass the input sentence through four binary relation extractors, each trained to detect one of the edge types. Each column represents the scores outputted by one binary relation extraction module. For example, the column labeled **bre-appeared** contains the “**appeared in**” edge scores for each “**from**” and “**to**” node pair. The higher the score, the more likely it is that an edge of that type with the corresponding “**from**” and “**to**” nodes exists. In the **bre-appeared** column, the highest score (0.866, in bold) occurs in the second row. This means that if an “**appeared in**” relation exists in the sentence, it is more likely that the edge points from the PAPERS node to the NIPS node, than any other combination of “**from**” and “**to**” nodes.

“from”	“to”	bre-affiliated	bre-appeared	bre-referenced	bre-wrote
PAPERS	VINYALS	-1.027	-0.318	-1.517	-0.693
PAPERS	NIPS	-3.031	0.866	-1.990	-2.083
VINYALS	PAPERS	-1.807	-1.956	-0.212	0.883
VINYALS	NIPS	-1.818	-0.689	-1.702	-1.106
NIPS	PAPERS	-2.887	-2.063	-1.128	-1.021
NIPS	VINYALS	-1.314	-1.725	-2.388	-1.413

Table 3.3: The inputs and outputs of the binary relation extraction models. The inputs are the trained NER-G model and a set of entity pairs. The output of each binary relation extractor is a set of scores associated with each combination of ordered endpoint nodes.

Binary relation extraction model

We use MITIE [15] to train the binary relation extraction models, which use the NER-G model described earlier for feature extraction and processing. MITIE offers several pre-trained models to detect relations such as author, organization founded, parents, place of birth, religion, nearby airports, ethnicity, nationality, people involved in an event, and more. We train binary relation extraction models specifically for the edge types defined in the property graph. Training from scratch allows the overall pipeline to generalize to different domains which may model different relationships.

3.3.4 Output re-organization

As a preprocessing step, we organize the NER outputs into a dictionary that keeps track of all of the nodes. The keys of the dictionary are the token's position in the sentence, and the values contain the node identifier and the attribute type (if applicable). We add each token marked as `N` by the NER-G model and each attribute detected by the NER-A model to the node dictionary. For our running example, the dictionary is $\{(3, 4): [\text{'n0'}], (6, 7): [\text{'n1'}, \text{'author_name'}], (9, 10): [\text{'n2'}, \text{'venue_name'}]\}$. In other words, the token located in the range $(3, 4)$ is assigned the node identifier `'n0'`, the token located in the range $(6, 7)$ is assigned the node identifier `'n1'` and attribute type `author_name`, and the token located in the range $(9, 10)$ is assigned the node identifier `'n2'` and attribute type `venue_name`.

3.4 Information processing phase

In this phase of the model pipeline, we organize information extracted from the named entity recognizers and the binary relation extractors into a structured format. The following sections (Node processing, Attribute processing, and Edge processing) correspond to the three sub-diagrams depicted in Figure 3-2b.

3.4.1 Node processing

For each token tagged by the NER-G module as `N`, the system creates a node structure. The node creator requires the node identifier, node type, and node attributes in order to create a node structure. The node identifier is obtained through a simple lookup in the node dictionary. The node type is obtained by comparing the token embedding to the embedding of each node type and choosing the node type with the closest embedding.

Word embeddings map tokens to vector representations such that semantically related words are located closer together in the vector space. Word embeddings avoid the curse of

dimensionality associated with other vector representations such as one-hot encoding, and provide a baseline approach for modeling word similarity. Several off the shelf embeddings are available for use today, such as Word2Vec [18] from Google and GloVe [22] from Stanford. We use pre-trained GloVe embeddings, which were obtained via unsupervised learning on the Wikipedia 2014 and Gigaword 5 [21] corpora.

In the example from Table 3.2, the NER-G module tags PAPERS as N. We compare the word embedding of PAPERS to the embeddings of the node types (“**author**,” “**paper**,” “**venue**,” and “**organization**”). The similarity scores of the word embedding of PAPERS to the embeddings of “**author**,” “**paper**,” “**venue**,” and “**organization**” are 0.496, 0.740, 0.131, and 0.345, respectively. The system suggests that the node type is “**paper**,” since the embedding of PAPERS is closest to the embedding of the node type “**paper**.” The final node structure produced by the system is:

```
{ 'id': 'n0',  
  'type': 'papers',  
  'attributes': [] }
```

3.4.2 Attribute processing

The system creates an attribute structure for each token tagged by the NER-A module. The attribute creator requires the attribute type and attribute value in order to create an attribute structure. The attribute type associated with the token is obtained from the node dictionary. From this information we know the node type, since each attribute type is mapped to one node type. If a node structure of the same node type already exists, the attribute is added to that node. Otherwise, a new node structure is created with the node type, attribute structure, and node identifier (which is also obtained from the node dictionary).

In the example from Table 3.2, VINYALS and NIPS are tagged by the NER-A module. In an earlier pre-processing step, these tokens were added to the node dictionary along

with their attribute type, and each was assigned a node number. From the node dictionary, we know that VINYALS is associated with node identifier `n1` and attribute type “author_name,” and that NIPS is associated with node identifier `n2` and attribute type “venue_name.” From this information we conclude that the node types associated with VINYALS and NIPS are “**author**” and “**venue**,” respectively. No node structures of type “**author**” or type “**venue**” currently exist, so the system creates a new node structure for each token. The final node structures produced by the system in this step are:

```
{ 'id': 'n1',  
  'type': 'author',  
  'attributes': [{ 'name': 'author_name', 'value': 'Vinyals',  
                  'op': '=' } ] }
```

```
{ 'id': 'n2',  
  'type': 'venue',  
  'attributes': [{ 'name': 'venue_name', 'value': 'NIPS',  
                  'op': '=' } ] }
```

3.4.3 Edge processing

We first compile a list of every token that was tagged as N or A by the NER-G model. Each binary relation extractor scores every combination of two tokens from this list. For each edge detected by the binary relation extractors (node pairs with a positive edge score), the system creates an edge structure. The edge creator requires the edge type, “**from**” node identifier, and “**to**” node identifier, which are determined via node dictionary lookups, in order to create an edge structure. If the information is not contained in the node dictionary, a new node is created with a new node identifier, node type given by the edge endpoint type requirements, and an empty attribute structure.

Continuing with the example “SHOW ME ALL PAPERS CO-AUTHORED BY VINYALS APPEARING IN NIPS,” the edges with positive scores in Table 3.3 are “**appeared in,**” from PAPERS to NIPS, and “**wrote,**” from VINYALS to PAPERS. The system obtains the node numbers corresponding to the PAPERS, VINYALS, and NIPS nodes, which are ‘n0,’ ‘n1,’ and ‘n2,’ respectively. The final edge structures produced by the system in this step are:

```
{ 'type': 'appeared', 'from': 'n0', 'to': 'n2' }
{ 'type': 'wrote', 'from': 'n1', 'to': 'n0' }
```

System output

In summary, the system translates the example natural language query, “SHOW ME ALL PAPERS CO-AUTHORED BY VINYALS APPEARING IN NIPS,” into the following structured subgraph query. As shown in Figure 3-2b, the output of the node processing step and the attribute processing step are node structures, and the output of the edge processing step is an edge structure. The system creates the structured query by aggregating all of the node structures and edge structures created in the previous steps of the model pipeline.

```
`nodes' :
[
  { 'id': 'n0',
    'type': 'papers',
    'attributes': [] },
  { 'id': 'n1',
    'type': 'author',
    'attributes': [{ 'name': 'author_name', 'value':
      'Vinyals', 'op': '=' } ] },
```

```

    {'id': 'n2',
     'type': 'venue',
     'attributes': [{'name': 'venue_name', 'value': 'NIPS',
                      'op': '='}]
    ],

'edges':
[
    {'type': 'appeared', 'from': 'n0', 'to': 'n2'},
    {'type': 'wrote', 'from': 'n1', 'to': 'n0'}
]

```

3.5 Post processing phase

3.5.1 User feedback and cleanup

Noise and error may arise at any step of this pipeline. Therefore, the user performs a final step of human review to resolve any ambiguities that the system cannot deal with. Given that the property graph format outputted by the pipeline is relatively interpretable by the average person, we expect that the user can correct any errors that are present. A user who wishes to query a graph database no longer bears the responsibility of recalling details regarding specific query languages. The final structured subgraph query is easily converted into a variety of graph query languages commonly used today, such as Gremlin. Therefore, our system acts as an intermediate representation that is not specific to any single domain, and can generalize to other datasets and query languages.

In the next section, we describe the performance of each submodule as well as the prediction accuracy of the system as a whole. We also present an approximation of the

number of edits necessary for a user to perform in this post processing phase in order to obtain the correct final structured language query.

Chapter 4

Results

4.1 Dataset

Table 4.1 presents the amount of training data available for each of the six modules used in the pipeline. We created 218 queries of varying complexity related to the graph database. The sentences range from a minimum length of 3 tokens to a maximum length of 24 tokens. Examples of sentences are “SHOW ME ALL PAPERS CO-AUTHORED BY VINYALS APPEARING IN NIPS” and “PAPERS WRITTEN BY PEOPLE AFFILIATED WITH MICROSOFT RESEARCH THAT ARE REFERENCED BY ARTICLES WRITTEN BY PEOPLE AFFILIATED WITH STANFORD.”

The number of graph component entities (entities tagged as N, E, or A) range from 301 to 415, and in total, there are 1035 such tags. The number of attribute type entities (entities tagged as organization name, venue date, paper title, venue name, organization location, venue location, paper year, or author name) range from 31 to 86. In total, there are 414 such tags.

Of the 218 sentences, 60 contain pairs of nodes with an “**appeared in**” relation, 61 contain pairs of nodes with an “**affiliated with**” relation, 61 contain pairs of nodes with a “**referenced**” relation, and 109 contain pairs of nodes with a “**wrote**” relation. The total

number of tagged pairs for each relation ranges from 556 pairs to 1344 pairs.

NER-G	
number of training sentences	218
average sentence length	10.07
number of (N) tags	301
number of (E) tags	319
number of (A) tags	415
total number of tagged entities	1035
NER-A	
number of training sentences	218
average sentence length	10.07
number of (org. name) tags	49
number of (venue date) tags	46
number of (paper title) tags	86
number of (venue name) tags	57
number of (org. location) tags	31
number of (venue location) tags	36
number of (paper year) tags	31
number of (author name) tags	78
total number of tagged entities	414

BRE-appeared	
number of training sentences	60
number of positive pairs	63
number of negative pairs	697
total number of tagged pairs	760
BRE-affiliated	
number of training sentences	61
number of positive pairs	69
number of negative pairs	803
total number of tagged pairs	872
BRE-referenced	
number of training sentences	61
number of positive pairs	63
number of negative pairs	493
total number of tagged pairs	556
BRE-wrote	
number of training sentences	109
number of positive pairs	132
number of negative pairs	1212
total number of tagged pairs	1344

Table 4.1: The amount of training data available for each named entity recognition (NER) module and each binary relation extraction (BRE) module.

4.2 Information extraction phase evaluation

We evaluate the performance of each module in the pipeline by computing precision, recall, and F1 scores. These metrics are defined below, where TP , TN , FP , and FN denote the number of true positives, true negatives, false positives, and false negatives, respectively.

$$precision = \frac{TP}{TP + FP} \quad recall = \frac{TP}{TP + FN}$$

$$F1 = \frac{2 * precision * recall}{precision + recall}$$

Precision is the fraction of retrieved elements, or instances predicted true, that are relevant. Recall is the fraction of relevant elements, or instances that are actually true, that were retrieved. F1 is a metric that combines precision and recall. Table 4.2 presents the results for the two named entity recognition and four binary relation extraction modules. For each model, we run k-fold cross validation by partitioning the training set into $k = 5$ splits. The values reported are the average of the scores when holding one split out for evaluation and using the rest of the data for training the model. For the NER models, we use all 218 sentences. Since the number of sentences for each BRE model varies, we randomly choose 60 sentences for each so that the scores are comparable.

model	precision	recall	F1
NER-G	0.861	0.869	0.865
NER-A	0.873	0.871	0.872
BRE-appeared	0.857	0.842	0.848
BRE-affiliated	0.897	0.819	0.857
BRE-referenced	0.783	0.720	0.750
BRE-wrote	0.860	0.845	0.853

Table 4.2: K-fold cross validation scores for each model (k=5).

The precision, recall, and F1 scores surpass 0.8 for all models except BRE-referenced. The complexity of the edge type explains some of the difference in performance. The **“referenced”** binary relation extractor performs worse because there is more ambiguity in determining the edge endpoint node types. In natural language, an author can reference another author, a paper can reference another paper, an author can reference a paper, or a paper can reference an author. For other edge types, the endpoint node types do not vary as much. As an example, the source node of the **“wrote”** edge is always an author node, and the target node is always a paper node.

To evaluate the effect of the amount of training examples on model performance, we repeat the same k-folds cross validation procedure as above, with varying dataset sizes. The resulting learning curves are shown in Figures 4-1 through 4-6.

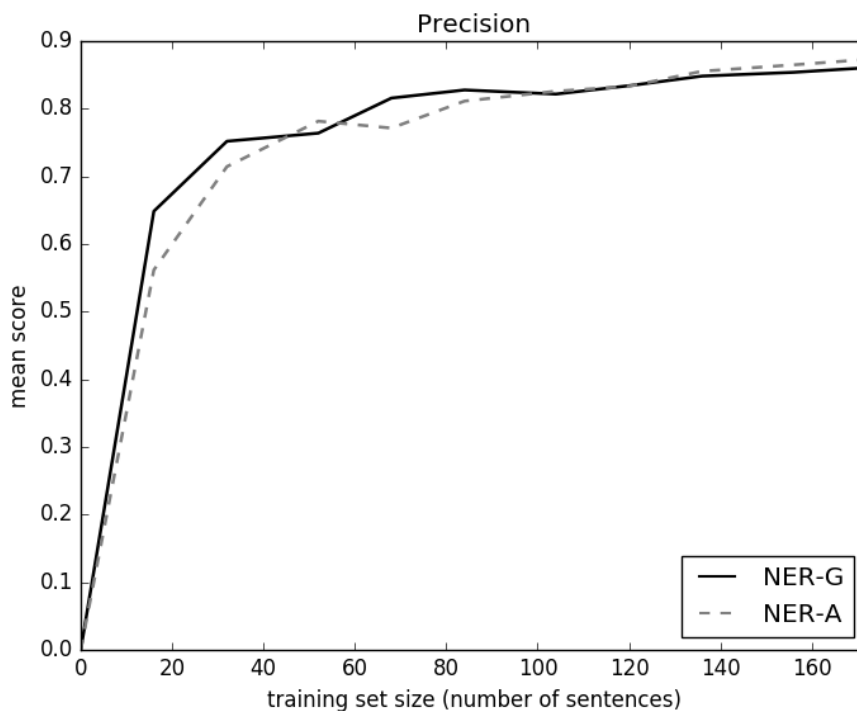


Figure 4-1: Average precision for the NER models using various amounts of training data.

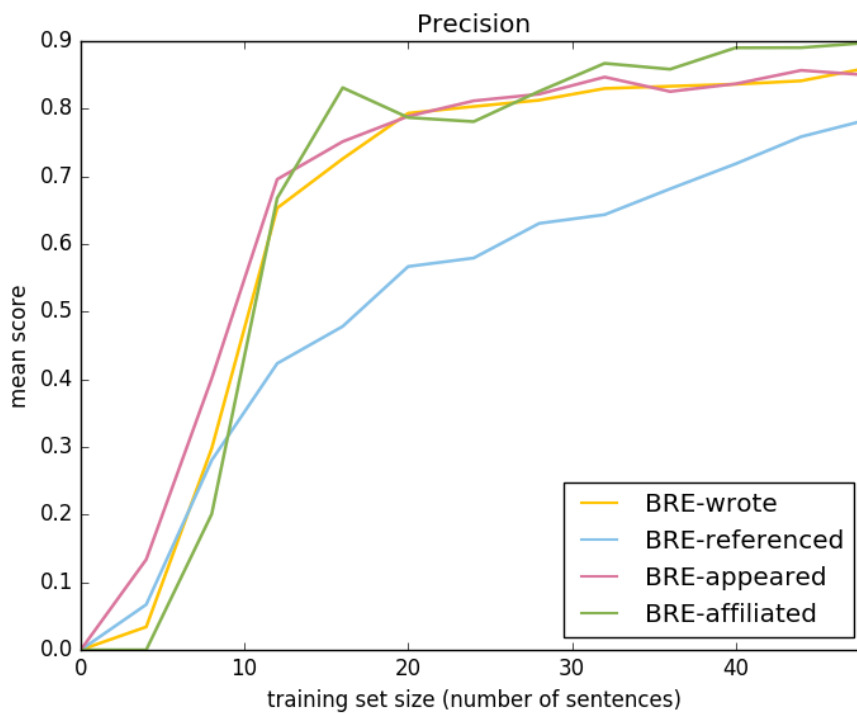


Figure 4-2: Average precision for the BRE models using various amounts of training data.

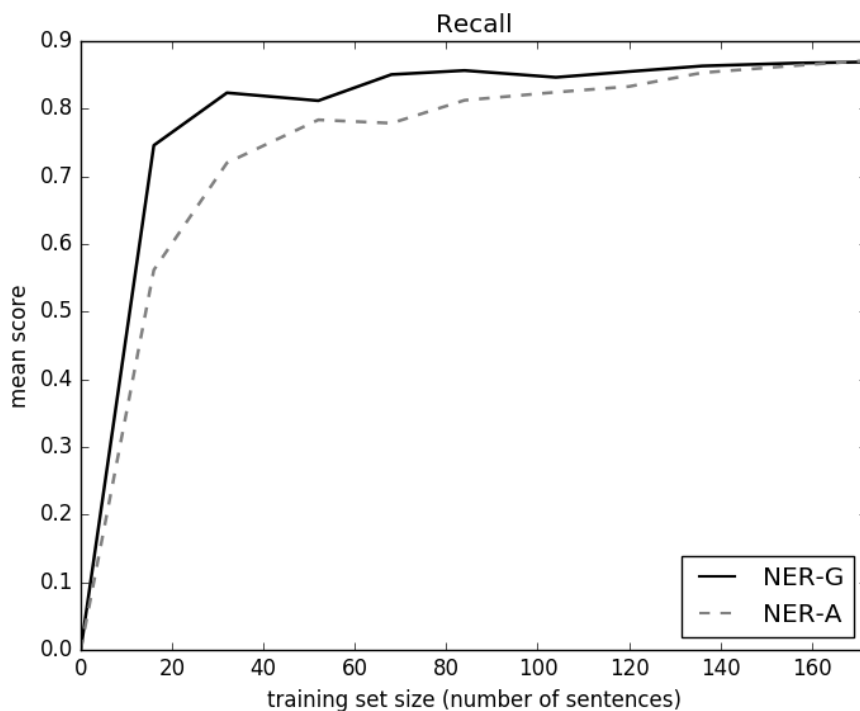


Figure 4-3: Average recall for the NER models using various amounts of training data.

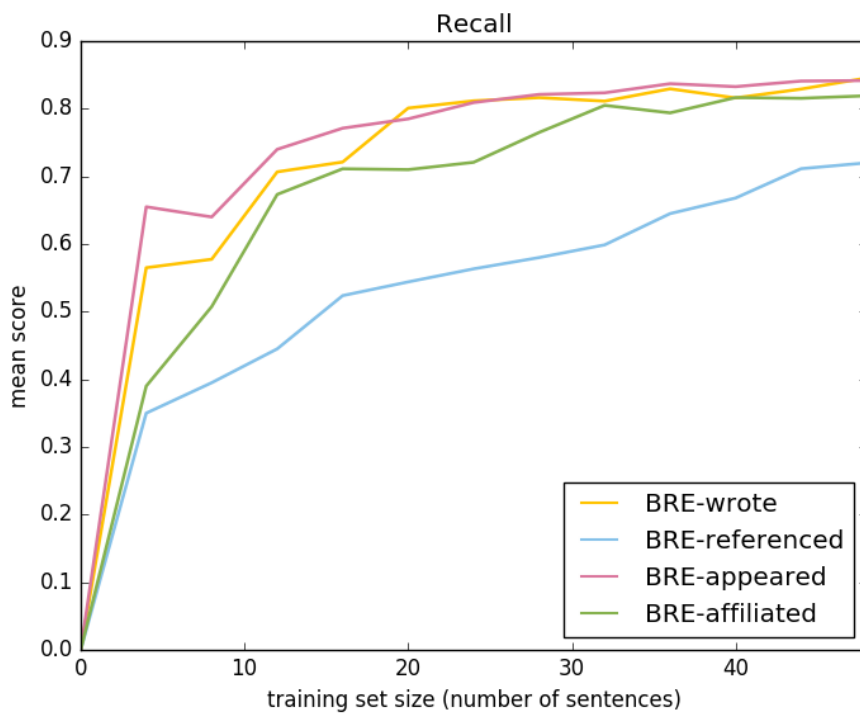


Figure 4-4: Average recall for the BRE models using various amounts of training data.

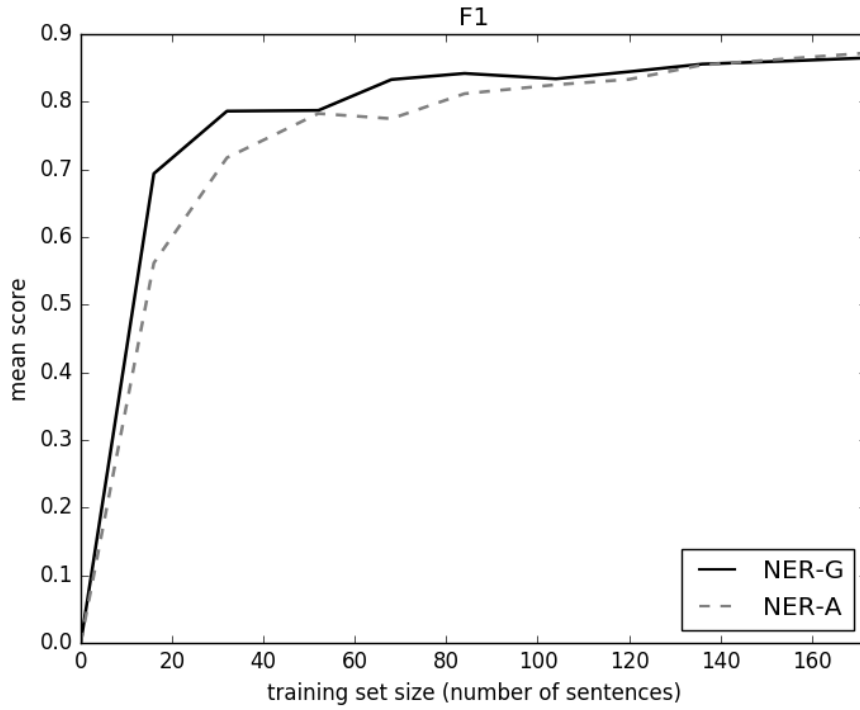


Figure 4-5: Average F1 score for the NER models using various amounts of training data.

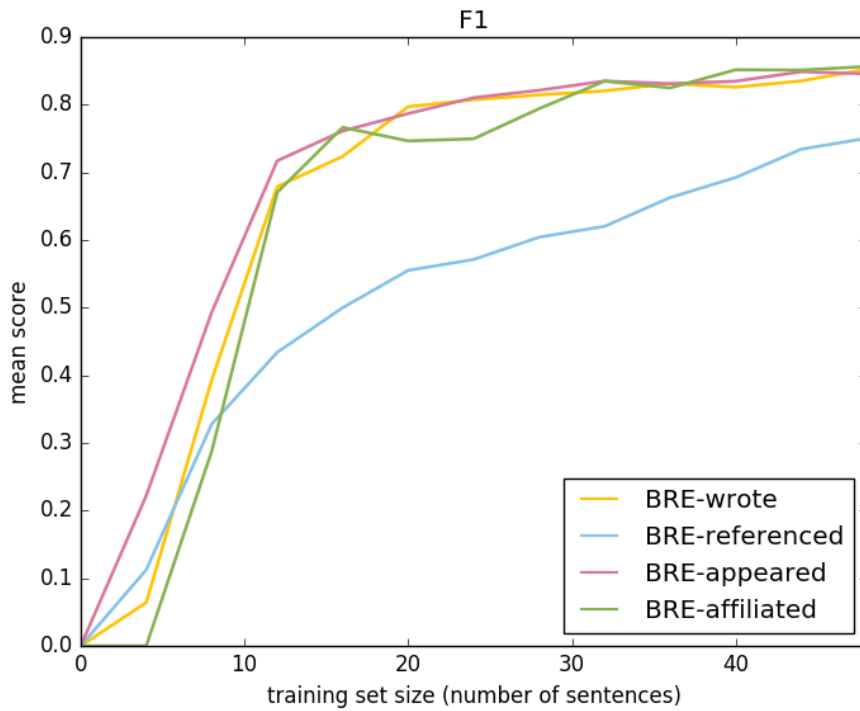


Figure 4-6: Average F1 score for the BRE models using various amounts of training data.

For all models and metrics, the mean score increases as the size of the training set increases. The curves plateau fairly quickly, and this suggests that a relatively small training set is sufficient. This could be due to the lack of variety in the way the graph components are expressed in the training set. Spending more resources on obtaining additional data may not be necessary, depending on the system’s accuracy requirements. As discussed previously, all scores surpass 0.8 with the exception of the BRE-referenced model.

4.3 Information processing phase evaluation

In order to evaluate the final system output, we compare the predicted structured language queries to the correct structured language queries. We compute the precision, recall, and F1 scores for predicting node structures and for predicting edge structures. We also report overall accuracy, which is the fraction of sentences whose predicted output exactly matches the correct structured language query, and an edit cost metric, which is described next. In the following sections, “predicted” refers to the outputs of the models and pipeline, and “gold” refers to the correct answer that the outputs are compared to.

We calculate a version of graph edit distance, a measure of the similarity between two graphs, to approximate the amount of work relative to the size of the subgraph query that a user would need to perform due to cases that the model pipeline cannot handle. In order to do so, we count the number of fields in the predicted structured language query that a user would need to edit in the post-processing step in order to obtain the gold structured language query.

We compute the relative cost by dividing the edit cost by the total cost of building the graph. To calculate the edit cost, we count the number of times a gold edge does not appear in the predicted edges (an edge must be added), the number of times a predicted edge does not appear in the gold edges (an edge must be deleted), the number of times a gold node does not appear in the predicted nodes (an edge must be added), and the number of times a

predicted node does not appear in the gold nodes (a node must be deleted). For each of these four cases, we add three points to the edit cost, since three fields in the structured language query are incorrect (for edge structures, the “to” node, “from” node, and edge type, and for node structures, the node ID, type, node attributes). The total cost is computed by adding one point for each node, attribute, and edge field in the gold structured language query. The reported edit costs may overestimate the true cost because edits usually involve swapping out individual fields rather than entire node or edge structures.

To quantify the effect of each module in the pipeline on the final structured prediction, we perform several experiments, in which the gold tags are used for some submodules, and the predicted tags are used for others. First, we simulate that every module makes predictions with perfect accuracy by using only the gold tags. In this case, any errors that occur must arise during the second phase of the model pipeline. We also test using the gold tags for all modules except for one module of interest. Second, we use the predicted tags for all submodules in the pipeline. We also report the scores when using the gold tags for one module of interest and the predicted tags for every other module.

	Nodes			Edges			Total	
gold/pred.	prec.	recall	F1	prec.	recall	F1	acc.	cost
GGGGGG	0.861	0.862	0.858	0.746	0.710	0.720	0.711	0.203
PGGGGG	0.821	0.834	0.823	0.702	0.667	0.676	0.646	0.276
GPGGGG	0.752	0.771	0.755	0.680	0.641	0.651	0.576	0.380
GGPGGG	0.832	0.850	0.835	0.677	0.680	0.665	0.655	0.279
GGGPGG	0.856	0.862	0.854	0.698	0.664	0.669	0.697	0.236
GGGGPG	0.785	0.814	0.788	0.592	0.639	0.593	0.590	0.432
GGGGGP	0.844	0.866	0.848	0.664	0.653	0.646	0.665	0.272

Table 4.3: The effect of each model in the information extraction phase on the structured language prediction scores (precision, recall, and F1 for nodes and edges, overall query accuracy and edit cost).

gold/pred.	Nodes			Edges			Total	
	prec.	recall	F1	prec.	recall	F1	acc.	cost
PPPPPP	0.687	0.727	0.695	0.449	0.511	0.453	0.469	0.684
GPPPPP	0.691	0.742	0.702	0.447	0.511	0.451	0.488	0.693
PGPPPP	0.749	0.790	0.757	0.456	0.521	0.461	0.534	0.595
PPGPPP	0.693	0.728	0.699	0.485	0.534	0.484	0.479	0.632
PPPGPP	0.687	0.727	0.695	0.475	0.541	0.483	0.465	0.668
PPPPGP	0.713	0.744	0.720	0.531	0.547	0.521	0.520	0.527
PPPPPG	0.688	0.727	0.695	0.482	0.540	0.486	0.474	0.650

Table 4.4: The effect of each model in the information extraction phase on the structured language prediction scores (precision, recall, and F1 for nodes and edges, overall query accuracy and edit cost).

The results are presented in Tables 4.3 and 4.4. The first column in each row denotes whether the gold tags (G) or the predicted tags (P) are used for that module, and the order of the modules is NER-G, NER-A, BRE-appeared, BRE-affiliated, BRE-referenced, and BRE-wrote. For example, PGPPPP denotes that the gold tags were used for the NER-A module, and the predicted tags were used for all of the other modules. Table 4.3 presents the results in the case where all models in the information extraction phase of the pipeline except one (or zero) use the gold annotations and Table 4.4 presents the results in the case where all models except one (or zero) use the predicted outputs.

The accuracy of the pipeline under the GGGGGG setting, which simulates perfect NER and BRE models, is 0.711. The edit cost to total construction cost ratio is 0.203. This means that on average, the user must edit approximately one fifth of the overall structured language query in order to obtain the correct answer. Given that the average structured language query in the dataset consists of 4.4 graph structures (2.7 nodes and 1.7 edges), this amounts to correcting fewer than one node or edge for each prediction. Since we assume in this setting that the models predict the graph components, attribute types, and edge relations with perfect accuracy, the error can be attributed solely to limitations in the information processing phase of the pipeline.

In reality, the user must deal with errors in earlier stages of the pipeline as well. The prediction accuracy under the PPPPPP setting, which uses no gold annotations for any of the models, is 0.469. The edit cost ratio is 0.684, which is equivalent to approximately 3 node or edge edits to the structured query prediction. As a reference point, we compare the prediction accuracy score of our model to the logical form accuracy achieved by the Seq2SQL [30] model. The results are not directly comparable due to differences in the dataset, target database type, and model. However, the task is similar in that both models attempt to translate natural language queries into structured formats for querying databases. Our model achieves an accuracy score of 0.469 using a dataset of only 218 training sentences. In comparison, the Seq2SQL model achieves a logical form accuracy of 0.483 using the WikiSQL dataset which consists of 80,654 pairs.

In the other settings of using gold or predicted annotations for each model, the precision, recall, and F1 scores are fairly similar within each table. In Table 4.3, we see that the edge type that results in the largest drop in performance when using the predicted annotations instead of the gold annotations is the **“referenced”** edge. In Table 4.4, the edge type that results in the largest gain in performance when using the gold annotations instead of the predicted annotations is also the **“referenced”** edge. These results are in line with the analysis of the individual models in Section 4.2. The accuracy of the NER-A model has a large effect on the structured language prediction scores, as well.

4.4 Qualitative analysis

Table 4.5 demonstrates how an example natural language query is represented at various stages of the pipeline. From manually reviewing the structured language predictions, we uncover sentence patterns that the system has trouble dealing with. Misclassified outputs of the named entity recognition and binary relation extraction modules contribute to some of the overall structured language prediction error. Many errors arise when combining these

outputs in the second phase of the model pipeline.

4.4.1 Word embeddings

Using word embeddings alone to determine the corresponding node type of each entity is inaccurate in many cases. For example, this strategy maps “PEOPLE”, “RESEARCHERS” and “WHO” to organization nodes instead of author nodes, “CONFERENCE” to an organization node rather than a venue node, and “COMPANY” to a paper node instead of an organization node. For the five examples above, we hard code the correct node type rather than compare embeddings. Instead of using word embeddings, the system could use a multiclass classifier to predict the node type of the entity.

4.4.2 Multiple attributes

The current system creates one node for each attribute and each node detected. Most of the time, these attributes are associated with the same node. Therefore, an effective node combination step is necessary in order to reduce prediction error. The system’s default behavior is to combine all nodes of the same type, and this is not always correct. For example, in the sentence “WHICH CONFERENCES WERE HELD IN TOULON, FRANCE IN 2017 AND SAN JUAN, PUERTO RICO IN 2016?”, all attributes detected belong to **venue** nodes. The system aggregates them into one node, rather than creating two separate **venue** nodes. Instead, it could use a binary relation extractor to determine whether or not two nodes relate to the same entity and should therefore be combined.

4.4.3 Implicit nodes and edges

In many cases, nodes are not explicitly stated in the natural language query. Therefore, the named entity recognizer has no way to tag them. In the example query “ARTICLES APPEARING IN ICML 2017 FROM OPENAI,” there is one implicit node and two implicit

Input	“SHOW ME ALL ARTICLES IN NIPS WRITTEN BY SOMEONE FROM GOOGLE BRAIN.”
NER-G	N: ARTICLES E: IN A: NIPS E: WRITTEN BY N: SOMEONE E: FROM A: GOOGLE BRAIN
NER-A	venue name: NIPS organization name: GOOGLE BRAIN
BRE-affiliated	(SOMEONE, GOOGLE BRAIN)
BRE-appeared	(ARTICLES, NIPS)
BRE-referenced	
BRE-wrote	(SOMEONE, ARTICLES)
Output	{‘NODES’: [{‘ATTRIBUTES’: [], ‘TYPE’: ‘PAPER’, ‘ID’: ‘N0’}, {‘ATTRIBUTES’: [{‘NAME’: ‘VENUE_NAME’, ‘VALUE’: ‘NIPS’, ‘OP’: ‘=’}], ‘TYPE’: ‘VENUE’, ‘ID’: ‘N1’}, {‘ATTRIBUTES’: [], ‘TYPE’: ‘AUTHOR’, ‘ID’: ‘N2’}, {‘ATTRIBUTES’: [{‘NAME’: ‘ORGANIZATION_NAME’, ‘VALUE’: ‘GOOGLE BRAIN’, ‘OP’: ‘=’}], ‘TYPE’: ‘ORGANIZATION’, ‘ID’: ‘N3’},], ‘EDGES’: [{‘TO’: ‘N1’, ‘FROM’: ‘N0’, ‘TYPE’: ‘APPEARED’} {‘TO’: ‘N0’, ‘FROM’: ‘N2’, ‘TYPE’: ‘WROTE’} {‘TO’: ‘N3’, ‘FROM’: ‘N2’, ‘TYPE’: ‘AFFILIATED’}] }

Table 4.5: The output of each submodule in the pipeline and the final structured language prediction for an example natural language input query.

edges. An author wrote the article, and the author is affiliated with OpenAI. To reduce prediction error, the system can make use of additional information such as the fact that

the source and target node types corresponding to an edge are predefined. For example, if the named entity recognizer for graph components detects that “FROM” corresponds to an **“affiliated with”** edge, the system could deduce from the schema that an author node and an organization node must be present and create the nodes if they are not.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Graph databases are useful for representing relationships between various entities. In recent years, the amount of knowledge stored in graph databases has increased significantly. This thesis presents a method for converting natural language queries into structured representations that are easily applied to graph databases. Many previous works include systems that convert natural language queries to SQL queries, which are then applied to relational databases. However, our contributions focus on converting natural language queries to subgraph queries.

We present a model pipeline consisting of an information extraction phase and an information processing phase. The first phase is composed of several named entity recognition and binary relation extraction models. The second phase applies an algorithm to combine the outputs gathered from the previous phase into structured queries.

When trained using all available sentences, the named entity recognition modules for graph components and for attribute types (NER-G and NER-A) achieve F1 scores of 0.865 and 0.875. The binary relation extractors achieve F1 scores of 0.848, 0.857, 0.747, and 0.861 for the “appeared,” “affiliated,” “referenced,” and “wrote” edges. Adding more high

quality training examples would improve the performance of these individual modules, resulting in higher prediction accuracies for the final pipeline output.

Zhong et al. [30] present a sequence to sequence model (Seq2SQL) that translates natural language queries into SQL queries, and report a logical form accuracy of 48.3%. Our model achieves an accuracy of 46.9%. The metrics reported are not directly comparable, since they involve queries for two different types of databases containing differing types of information. Furthermore, our dataset contains 218 training sentences, whereas the Wik-iSQL dataset contains more than 80,000. We report the Seq2SQL score as a reference point because the task is similar to ours. One reason that we are able to achieve a similar score despite using a small dataset is that our model breaks the task down into several subcomponents, whereas the Seq2SQL model is more end to end.

5.2 Future work

Incorporating user feedback

Further human evaluation may be conducted by deploying the system to a wider audience. Since the system may produce incorrect results, an extension could be to implement a simple user interface that allows users to make corrections more easily.

In addition, the system requires a training set of sentences. Initially, this training set may be limited in number. It could incorporate user feedback and retrain the model periodically by adding previously asked queries to the training set. This would create a richer, more diverse set of questions for the submodules to learn from, since each user has their own style of question-asking.

Detecting other graph components

The current named entity recognizer for graph components recognizes nodes, attributes, and edges. Depending on the domain in which the system is deployed, it may be useful to

include other graph components. For example, the system could learn to detect temporal and spatial entities (such as a period of time or a distance) and relations (less than, greater than, equal to, etc.) so that a user could query a database for events that were located within a certain distance of a point of interest, or events that occurred within a certain time frame.

Improving ambiguous node type predictions

The success of using word embeddings to predict node type depends on how similar the entities are to the node types specified by the graph. For example, the system may think that Stanford refers to the name of a person, rather than the name of an organization. The system could search the graph for various tokens in order to determine if they commonly appear in the database as person, venue, paper, or organization names. This preliminary search could help to resolve such ambiguities. Another possible solution is to train a multiclass classifier which predicts the node type from the node entity. To address edge types whose endpoint node types are ambiguous in natural language, such as the “**referenced**” edge, the system could relax the requirement that the endpoint nodes must be of a fixed type.

Other future work

The system pipeline could make use of other natural language processing tools. One of the main challenges that the system currently faces is when to combine nodes of the same type, and when to keep them separate. An off the shelf coreference resolution model could be used to combine nodes that refer to the same entity. Another possible solution is to train a new binary relation extractor as mentioned in Section 4.2.2.

Appendix A

Pseudocode for model pipeline

1. Train named entity recognizer for graph components (NER-G)
2. Train named entity recognizer for attribute types (NER-A)
3. For each edge type:
 - Train binary relation extractor (BRE-edge-type) given NER-G
4. Use NER-G to predict 'N', 'E', 'A' entities given input sentence
5. For each entity tagged 'N' by NER-G:
 - Record in node_dict (key = entity position, value = node id)
6. Use NER-A to predict attribute tokens and their attribute type given input sentence
7. For each entity tagged by NER-A:
 - Record in node_dict (key = entity position, value = node id, attribute type)
8. For each entity tagged 'N' by NER-G:
 - Get node type through embedding comparison
 - Get node id through node_dict lookup
 - Create node structure, add it to overall

```

        structure
9. For each entity tagged by NER-A:
    - Get attribute type through node dictionary lookup
    - Create attribute structure
    - Get node type from attribute type
    If a node structure of the same type exists:
        - Add attribute structure to that node
    Else:
        - Get node id through node_dict lookup
        - Create a new node structure
        - Add attribute structure to the new node
        - Add node structure to the overall structure
10. Get relation_pairs:
    For source_node in node_dict:
        For target_node in node_dict:
            If source_node != target_node:
                relation_pair = (source_node, target_node)
11. For each relation_pair in relation_pairs:
    For each edge type:
        Use BRE-edge-type to predict edge_score
        If edge_score > 0: #relation exists
            - Get source node id through node_dict look-
              up or create new node if it is not present
            - Get target node id through node_dict look-
              up or create new node if it is not present
            - Create edge structure, add it to overall
              structure
12. Present final structure to user for revision

```

Bibliography

- [1] Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. MASQUE/SQL: An efficient and portable natural language query interface for relational databases. In *Proceedings of the 6th International Conference on Industrial and Engineering Applications or Artificial Intelligence and Expert Systems*, pages 327–330, 1993.
- [2] Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. Natural language interfaces to databases - an introduction. *Natural Language Engineering*, 1:29–81, 1995.
- [3] Florin Brad, Radu Cristian Alexandru Iacob, Ionel Alexandru Hosu, and Traian Rebedea. Dataset for a neural natural language interface for databases (NNLIDB). In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 906–914, Taipei, Taiwan, November 2017. Asian Federation of Natural Language Processing.
- [4] Nick Cercone, Paul Mcfetridge, Fred Popowich, Dan Fass, Chris Groeneboer, Gary Hall, and Dan Fass Chris Groeneboer. The SystemX natural language interface: Design, implementation and evaluation. Technical report, Centre for Systems Science, Simon Fraser University, Burnaby, British Columbia, 1994.
- [5] Duen Horng Chau, Christos Faloutsos, Hanghang Tong, Jason I. Hong, Brian Gallagher, and Tina Eliassi-Rad. GRAPHITE: A visual query system for large graphs. In *Proceedings of the 2008 IEEE International Conference on Data Mining Workshops*, pages 963–966, Washington, DC, USA, 2008. IEEE Computer Society.
- [6] Paramveer S. Dhillon, Dean P. Foster, and Lyle H. Ungar. Eigenwords: Spectral word embeddings. *Journal of Machine Learning Research*, 16(1):3035–3078, 2015.
- [7] The Apache Software Foundation. Apache Tinkerpop, 2018.
- [8] Gary G. Hendrix, Earl D. Sacerdoti, Daniel Sagalowicz, and Jonathan Slocum. Developing a natural language interface to complex data. *ACM Transactions on Database Systems*, 3(2):105–147, 1978.
- [9] Ho Hoang Hung, Sourav S. Bhowmick, Ba Quan Truong, Byron Choi, and Shuigeng Zhou. QUBLE: Blending visual subgraph query formulation with query processing on large networks. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1097–1100. ACM, 2013.

- [10] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 963–973, Vancouver, Canada, July 2017. Association for Computational Linguistics.
- [11] Nandish Jayaram, Rohit Bhoopalam, Chengkai Li, and Vassilis Athitsos. Orion: Enabling suggestions in a visual query builder for ultra-heterogeneous graphs. *CoRR*, abs/1605.06856, 2016.
- [12] Changjiu Jin, Sourav S Bhowmick, Byron Choi, and Shuigeng Zhou. PRAGUE: Towards blending practical visual subgraph query formulation and query processing. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering Workshops*, pages 222–233, April 2012.
- [13] Changjiu Jin, Sourav S Bhowmick, Xiaokui Xiao, James Cheng, and Byron Choi. GBLENDER: towards blending visual query formulation and query processing in graph databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 111–122, June 2010.
- [14] Thorsten Joachims, Thomas Finley, and Chun-Nam John Yu. Cutting-plane training of structural SVMs. *Machine Learning*, 77(1):27–59, 2009.
- [15] Davis E. King. Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research*, 10:1755–1758, 2009.
- [16] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander Rush. OpenNMT: Open-source toolkit for neural machine translation. In *Proceedings of ACL 2017, System Demonstrations*, pages 67–72, Vancouver, Canada, 2017. Association for Computational Linguistics.
- [17] David L. Waltz. An english language question answering system for a large relational database. *Communications of the ACM*, 21:526–539, 1978.
- [18] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems*, volume 2, pages 3111–3119, 2013.
- [19] Chemseddine Nabti and Hamida Seba. A simple algorithm for subgraph queries in big graphs. *CoRR*, abs/1703.05547, 2017.
- [20] Inc. Neo4j. Intro to Cypher, 2018.
- [21] Robert Parker, David Graff, Junbo Kong, Ke Chen, and Kazuaki Maeda. English Gigaword fifth edition LDC2011T07. DVD, Philadelphia: Linguistic Data Consortium, 2011.

- [22] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics.
- [23] M.J.D. Powell. The BOBYQA algorithm for bound constrained optimization without derivatives. Technical report, Department of Applied Mathematics and Theoretical Physics, Centre for Mathematical Sciences, University of Cambridge, Cambridge, England, 2009.
- [24] Eric Prud’hommeaux and Andy Seaborne. SPARQL query language for RDF, 2013.
- [25] Paulo Reis, João Matias, and Nuno Mamede. Edite - a natural language interface to databases: A new dimension for an old approach. In *Information and Communication Technologies in Tourism 1997*, pages 317–326, 1997.
- [26] Marko A. Rodriguez. The Gremlin graph traversal machine and language. In *ACM Proceedings of the 15th Symposium on Database Programming Languages*, pages 1–10. ACM, 2015.
- [27] Hanghang Tong, Christos Faloutsos, Brian Gallagher, and Tina Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 737–746. ACM, 2007.
- [28] Joseph Weizenbaum. ELIZA - a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45, 1966.
- [29] William Woods, Ronald Kaplan, and Bonnie Webber. The Lunar science natural language information system: Final report. Technical report, Bolt Beranek and Newman Inc., Cambridge, Massachusetts, 1972.
- [30] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017.