STRUCTURED ENTITY QUERYING OVER UNSTRUCTURED TEXT

BY

JIAHUI JIANG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Associate Professor Kevin C. Chang

ABSTRACT

The web is a collection of unstructured webpages. This characteristic makes it very hard for users to search complex queries – in most of the times, queries are just plain text and it is very difficult for users to describe the internal relationships that they contain.

The EntityRank system allows users to specify the target entities for which they are interested within the query, which brings us one step closer to being able to describe the entity relationships. But this expressiveness is still limited because the queries are still in a flat format.

On the other hand, SQL queries for relational databases are very expressive. Users can easily specify multiple entities and the relationships between them. But in order to use SQL queries, we need to extract all the entities and relationships existing in the domain beforehand. The cost of maintaining the tables is very large. Also it is very hard if we want to add or modify the schema after the initial domain design.

Thus we want to build a system that can combine the advantages of the flexibility and informativeness of unstructured webpages and the expressiveness of SQL queries. In this work we design a system which allows users to use structured SQL queries on unstructured webpages using the help of the EntityRank system. We design the conceptual framework to map the concepts between two systems and also propose a ranking algorithm for the final results.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Problem Description

One of the limitations of the traditional search engine is that both the webpages and queries are flat and unstructured. When users are interested in one single entity, it is very hard for the system to detect what the target entity is and how to find this entity from unstructured webpages. So the returned results for the users will still be a collection of unstructured pages. Furthermore the users will have to navigate through all the pages to find out whether any one page contains the information they want. After seeing enough pages, the users still need to aggregate all the information they saw in order to get an idea of the overall importance of a certain entity.

In order to address this problem, an entity search system was designed [1, 2], where users can use entity tags to refer to the entities of interests. In this system, we will extract and index all the entities from a set of webpages beforehand. Then when given a query, we will look up into the index to find out all the entities that co-occur with other keywords inside the query and generate a table of ranked entities based on the score calculated for each occurrence.

Even with this entity search system, we can still find some scenarios where the system can't provide a satisfying result:

1. **Searching amidst complex internal relationships in the user's intuition:** for example, when we want to search for an online shop where we can buy a Canon 5D camera and also take advantage of free shipping for each purchase. Human beings can easily divide the query into two queries "Canon 5D online shop" and "free shipping online shop" and our targets are the intersection of the results from these two queries.

2. **Searching amidst many constraints listed in the query:** for example, when we want to find out all the lecturers that have taught in New York City, Chicago, San Francisco or Seattle. Here the constraint is a union of four shorter constraints - “all the lecturers that have taught in one particular city”.

Both of the issues are caused by the “flatness” of the query. “Flatness” means all the requirements have to be represented using one single sentence, where all the keywords are of equal importance and no extra structure of the query is provided. For the first scenario, because all the keywords are written in a flat format, it is hard for the search engine to understand which keywords are used as constraints for which keywords (entities). For the second scenario, since the query cannot be divided into sub-queries based on the entity it is related to, we can only try to find one single page matching all the information in the query. But in the real world, the information needed for one query might need to be collected from more than one page.

Thus, we consider providing users a query language that is more structured. SQL is widely used by a large number of users as a structure query language for relational databases. The structure of SQL queries can naturally represent multiple entities and their relationships inside. But the drawback is that in order to use SQL query we have to build the database first. Fully extracting all the relationships can be very time and space consuming. Since we do not know what tables users may want to use, we have to enumerate as many relationships as possible and store all the tuples satisfying the relationships physically. But no matter how many relationships we extract beforehand, users may always want to use a new relationship that is not yet extracted. So pre-extracting all the relationships and storing them is not only expensive but also not flexible.

Thus, we want to design a structured entity querying system for unstructured webpages which does not have a large space requirement.

Here we list two desired properties for designing the structured querying system over unstructured webpaegs:

1. Exact SQL syntax
   Since SQL is a language that a lot of users are familiar with, we want to provide a querying system where users can type a query that has exactly the same syntax and conceptual notions as SQL.

2. Minimal Extraction

   In order to reduce the space cost, we do not want to maintain any real table, which stores the relationship pairs. But in order to ensure efficiency, we still need to do some minimal extraction.

   By only using this basic extraction, we can achieve a trade-off between efficiency and space cost.

1.2 Challenges and Our Approach

For our problem, we want to design an entity querying system which achieves a trade-off between efficiency and the flexibility. We also need to consider the ranking problem because we only output the final tuples that satisfy the user's requirements based on many occurrences we see from the corpus. The rank of a tuple is the most direct way for users to understand the overall confidence of the current result they are looking at.

In this system, we decide to only maintain a minimal extraction by only extracting and building the inverted index for all the entities existing in the corpus. We also design a conceptual model that helps us to link together the structured SQL query with the back-end text searching system. Then a ranking model is provided to theoretically explain the confidence of each of our result tuples. We finally evaluate the system by checking its precision@k.

1.3 Organization of The Thesis

In this thesis, we propose a structured entity querying system over unstructured webpages, where we can provide users a structured query interface but only need to extract and store the minimum index. In Chapter 2, we will present the related work for the system. In Chapter 3, we will introduce both the conceptual model of the system and also the implementation design. In Chapter 4, the ranking model will be motivated and introduced. Then in Chapter 5, we will show the system demo and evaluate the search results of the query system. Finally in Chapter 6, we will provide the conclusions and discuss future work.

CHAPTER 2

RELATED WORK

The object of our work is to design an entity querying system for an unstructured data corpus which achieves a trade-off between efficiency and flexibility.

There are some entity search systems for unstructured data that have only considered the efficiency issue. In [3, 4, 5, 6], a database system is preconstructed by defining many possible domains and relationships for each domain and then instantiating each relationship from the web. After the database system is built, structured queries can be directly used on top of it.

There are also works done by only considering flexibility and minimal preprocessing. These systems [7, 8, 9, 10] achieve their goals by segmenting the input text query into short queries and then searching each of the queries separately. But since these systems do not do any extraction beforehand, both the intermediate results and the final outputs are in an unstructured page format. Therefore it is still not very convenient for the users to directly find out what they want.

Some other related systems [1, 2] allow users to use simple structured queries on unstructured data. For example, in the EntityRank system [1], users can write a text query that contains keywords and target entity types indicated by hashtags. One example query users can search is (#person United States president). This query implies that users are looking for a person entity that happens to be the president of United States. Users can also search for multiple entities inside one query or require an entity to have a certain value. When given a query, the entity rank system will look into the inverted index it built for all the entities and keywords and find all the occurrences of the target entity types that have occurred with the rest of keywords in the query. The final outputs are tuples of the values of the required entities ranked by the aggregated scores for each occurrence. But the limitation of this system is also very obvious. Assume we are given the following scenario, where users want to find all the students of Professor

Kevin Chang who are currently working at Yahoo. The entity search query users can write is "#person student of professor (#person = "Kevin Chang") works at (#organization = "Yahoo")". We will not get good results for this query due to two reasons:

1. When there are complex internal relationships in the query, since the query is flat, we cannot know which constraint keywords are used to describe the target student, which are used to describe the professor and which are used to describe the organization Yahoo.

2. When there are multiple constraints in the query, sometimes we may not be able to get all the information from just one single page. For instance, in our example the student list of Kevin Chang can only be found at Kevin Chang's homepage while the student's work history can only be found on his Linkedin pages. But since we do not have enough information to divide the query, we will not get any results if the above information does not appear together inside one page.

Another system [11, 12] called the CQS system is built upon the EntityRank system [1] we just described. Here an entity-oriented structured query language is designed for users to express complex relationships that can be used to search over unstructured webpages.

The typical syntax of a CQL query is:

```
SELECT #E1.val, #E2.val, ... , #Ek.val
FROM #entitytype_t1 AS #E1, #entitytype_t2 AS #E2, ... ,
        #entitytype_tk AS #Ek
WHERE pattern=("[#Ei keywords #Ej]<range>") AND ... AND
        pattern=("[#Et keywords]<range>")
GROUPBY #E1, ..., #Ek
```

For example, a CQL query that searches for all the people that are professors teaching at a university located in Illinois can be written as:

```
SELECT #p.val, #org.val
FROM #person AS #p, #organization AS #org
WHERE pattern=("[#p professor]<10>") AND pattern=("[#org
   university]<10>") AND pattern=("[#org located in Illinois
   ]<10>") AND pattern=("[#p teaches at #org]<10>")
GROUPBY #p, ..., #org
```

In the FROM statement, we list each entity of interest using a hashtag plus the entity type, for example, here we are interested in (professor, university) tuples, so in our FROM statement we select from #person and #organization. We can also create an alias for each different entity to make the representation easier (the #p and #org in our example).

In the SELECT statement, we can choose to print out the value of the target entities. This can be achieved by simply adding ".val" to the alias of the target entities.

The WHERE statement contains a list of patterns, where each pattern is also an EntityRank query. A pattern is used to describe what keywords we want each entity to co-occur with. For example, we want the person we are looking for to be a professor, so we construct one query as "#p professor". We can also use extra parameters, such as range, to describe how close we want these keywords to co-occur with the entity. For example a pattern of ("[#p professor]$< 10 >$") means we want the PERSON entity and the keyword professor to co-occur within a keyword window of size 10.

Then in the GROUPBY statement, users can specify 0 to k entities. This will group together all the tuples that have the same values for the required attributes. For example, if we group our results on both #p and #org, then in the final results, tuples with the same person entity value and organization entity value will only appear once. The ranking will be decided based on the aggregated score for all the occurrences that lead to this final tuple.

Even though this CQL system provides users a nice interface to use structured queries to search over unstructured webpages, it is conceptually unlike other structured query languages that users are familiar with.

For example, the most widely used structured query language is SQL, where given a domain, users can define relationships that they are interested in using a schema. A schema describes the structure of the relationship, for example, the name and type of the attributes in this relationship. If we consider the same professor-school-Illinois example as we used for CQL system, we can define two relationships here: TEACH and SCHOOL. Here the schema of the TEACH table tells us it has two attributes: professor_name and school_name. The schema of the SCHOOL table tells us it has two attributes: school_name and location. Then what we need to do when using a SQL query is simply to join the tuples from these two tables if they share the same professor name and if the school has its location as "Illinois." Tuples can only appear in the table if they satisfy the semantic meanings of the table. This design is relationship-oriented where the basic unit of the system the tuple, and all the semantic meanings are implied in the schema.

But for CQL queries, the whole design is entity-oriented. Users need to express what kind of entities they are looking for by constructing patterns. All the semantic meanings of an entity or between a pair of entities need to be specified using the keywords co-occurring with them.

CHAPTER 3

SYSTEM DESIGN

3.1 Query Language Syntax Design

3.1.1 Conceptual Design

Recall that our goal for the system is to let it achieve a balance between flexibility and efficiency. Our approach is to do a minimal extraction in the back-end, where we only extract and build an inverted index for the basic entities and keywords, and use a SQL querying interface for the front-end, which allows users to construct complex queries.

In order to use structured queries on unstructured webpages, we designed a conceptual model, which can map the concepts in a relational database to an unstructured dataset.

We use a school-professor example to explain our design, but our system is general and can be applied to any domain.

Without any knowledge about what domain the user may be interested in, our system always maintains the inverted index for basic entity types (PERSON, ORGANIZATION, LOCATION) and all other simple keywords. The indexes are built offline before user types any command. They will be used in finding the target entities after we know the input query.

Suppose the user is interested in a school-professor domain. If he is searching in a relational dataset, the tables that he may want to see can be described as follows, where the names of tables and the attributes of each table are listed below:

1. $T_{Professor}$

   Attributes: name (key), department

2. T_{School}

   Attributes: name (key), location

3. T_{Teach}

   Attributes: professor_name (composite key), school_name (composite key)

One important assumption we need to make is that we assume the name of the entity is also the key attribute. This means, for example, if we see two people with the same name, we assume they are the same real world person. We need this assumption because this system focuses on handling complex queries and we choose to consider the entity resolution problem as future work.

Also we only consider those functional dependencies between the key attributes and other attributes and ignore those dependencies that are only related to non-key attributes. For example, given a SCHOOL table with SCHOOL, CITY, ZIP as its attributes, based on the semantic meaning, the value of the CITY attribute is not only determined by the key attribute SCHOOL, but also by attribute ZIP. But in our system we choose to only consider the dependency between CITY and key attribute SCHOOL. This is because allowing users to define functional dependencies will make the interface overcomplicated for the users and we believe the functional dependencies between non-key attributes are not very common in most searching scenarios.

When we want to "instantiate" each table, we need some additional information from the user. This is because instead of having pre-built relational tables as in searching in a relational database, we want the system to be able to be used in any domain, and thus we want it to have the ability of constructing all the tables by itself from an unstructured corpus. For each key attribute, we require the user to provide its basic entity type and the keywords that it should co-occur with. We call those keywords as context keywords because they are used to provide more context to describe the entity. For example, in order to find all the possible values for the NAME column in $T_{Professor}$, the user needs to provide the information saying the professor name should be a PERSON entity and this PERSON entity should co-occur with another keyword, "professor," in the corpus. For those attributes that are not the only key attribute, in order to make their values satisfy the semantic meaning of the table, we also require them to co-occur with the key attribute or the other part of the composite key with the context keywords. Because in this case, simply satisfying the attribute's own semantic meaning is not enough. An attribute should also be related to the key entity with the correct relationship in order to appear in the table. For example, each possible value in the "professor_name" column in T_{Teach} needs to not only be a PERSON entity that co-occurs with another keyword "professor" but also needs to co-occur with a school name and a surrounding keyword such as "teach", which indicates that he should not only be a professor but also teaches at a school.

To make the process of defining attributes and context keywords for each table easier for the user, we define two standard types of tables the user can create: Specialized Entity and Relationship.

We use the word "specialized entity" to represent an entity type that is a subcategory of a basic entity type. Basic entity types include Person, ORGANIZATION and LOCATION. Some possible specialized PERSON types are: PROFESSOR, which is a subgroup of PERSON, or SCHOOL, which is a subgroup of ORGANIZATION. Each Specialized Entity table contains the information about one type of specialized entity and its attributes. The name of the specialized entity will be the only key of the table.

So in our system we assume that the only dependencies we have in Specialized Entity tables are from the key attribute to each non-key attribute.

In the above example, $T_{Professor}$ and T_{School} are both Specialized Entity tables.

Relationship tables contain the information about how entities are related to each other. Here we only consider two-way relationships because a three-way relationship table can be easily divided into three two-way relationship tables where each table describes the keyword co-occurrence requirements for each pair of entities. In every Relationship table, the names of the two entities will form the whole composite key together. Currently we do not allow a Relationship table to have extra attributes besides the two composite keys in order to keep the design and interface concise and easy to use.

In the above example, T_{Teach} is a Relationship table.

We list the whole design of the entity types and the context keywords as follows:

1. $T_{Professor}$

   name: type - PERSON, context - professor

   department: type - ORGANIZATION, context - department, context with the key attribute - work

2. T_{School}

   name: type - ORGANIZATION, context - school

   location: type - LOCATION, context - NULL, context with the key attribute - locate

3. T_{Teach}

   professor_name: type - PROFESSOR

   school_name: type - SCHOOL

   context: teach

One thing to notice is, in our system, we do not actually instantiate and store the content of each table. Instead, we only store the context requirements and use this knowledge to search for tuples after we are given the input query.

Here we use an example to explain how the whole system works.

Assume the user wants to find the names of all the professors working in schools located in Urbana. The SQL query he may write based on the tables we describe above is:

```
SELECT professor_name
FROM Teach, School
WHERE Teach.school_name = School.name
AND School.location = 'Urbana'
```

We list the constraints shown in the query and how the system can find the target values as follows:

1. We want to find out the names of professors. (Our targets are PERSON entities that have co-occurred with the keyword "professor".)

2. The professor needs to teach at an organization. (The PERSON name needs to co-occur with the ORGANIZATION and has the keyword “teach” between them.)

3. That organization should be a school. (The PERSON should appear near the keyword “school”.)

4. The school needs to be located in Urbana. (The ORGANIZATION has co-occurred with keywords “Urbana” and “locate” at the same time.)

Here the results for each constraint can be found using one entity search query as in the EntityRank system we described in Chapter 2. For example, in order to find all the PERSON entities that have co-occurred with the keyword “professor,” we just need to use the results returned by the entity search query “(#PERSON professor)”. In order to get the results that simultaneously satisfy multiple constraints, we can use CQL queries introduced at the end of Chapter 2, where we specify all our target entity types in the SELECT statement and then list all the entity search statements together into the WHERE statement.

Thus with the help of the conceptual mapping we just described, we are able to map a SQL query into a CQL query. From this CQL query, we can easily get our final results with the EntityRank system that has already been built.

3.1.2 Syntax Design

In this section, we want to describe the details about the grammar of the available queries that users can use.

In the query language, we supports two types of queries: (1) queries used to define the tables that the user wants (2) queries used to describe the constraints those target entities should satisfy. We discuss the syntax design for these two types of queries separately.

1. **Create Tables**

   Different from the normal SQL requirements, when a user wants to define a new table in our system, he needs to provide not only what attributes are used (containing the attribute names and attribute types), but also the semantic meaning of each attribute and the relationships between every pair of attributes (represented using context keywords). If we want to adapt the same CREATE TABLE syntax as in SQL queries, we can create an attribute called "context" for each table, which contains what keywords should be used to describe attributes. But since this context needs to be initialized when users create the table and also the value should be the same for every tuple in the table, the "context" attribute does not match the natural design of an attribute. So we have decided to use a question-answering format to ask the user what his requirements are for the new table.

There are two types of tables the users can create:

- **Specialized Entity**

  We defined a "specialized entity" as an entity that belongs to a subcategory of a "basic entity" in section 3.1.1. A specialized entity can have its own attributes. For example, a school can have a location attribute. Each attribute should have a name, a basic entity type and some context keywords used to describe its relationship with the specialized entity.

  The questions we need to ask the user when he wants to create a new type of specialized entity are:

  (a) What is the name of the new specialized entity?

  (b) What is the basic entity type behind it?

  (c) What is the name of the key attribute?

  (d) What are the context keywords that should be used to distinguish the basic entity type and current specialized entity type?

  (e) Do you want to add an attribute for the current entity?
  If so,

     i. What is the name of the attribute?

     ii. What is the basic type of the attribute?

     iii. What are the keywords that should be used to describe the relationship between the current attribute and the specialized entity?

Suppose the user is constructing the T_{school} table as in our example in section 3.1.1, the answers to these questions would be:

(a) School

(b) Organization

(c) Name

(d) School

(e) Do you want to add an attribute for the current entity?
Add one attribute

   i. Location

   ii. Location

   iii. Locate

We store all the answers into a local file. When we see the input query, we can use this information to translate the SQL query into a CQL query. We will discuss the formal translation algorithm in the next section. But an example we can see now is, if the user wants to find all the possible values for the NAME column in table SCHOOL, we can look up into our local file and see whether there is a table called SCHOOL. If so, we will see what the basic type of NAME column and what context keywords we need when looking for it. Finally we will be able to construct the entity search query as (#ORGANIZATION school).

- **Relationship**

  Users can also create relationship tables. Here we only allow users to create two-way relationship tables.

  The questions we need to ask the user when he wants to create a new type of relationship are:

  (a) What is the name of the relationship?

  (b) What is the name of the first entity in this relationship?

  (c) What is the type of the first entity in this relationship?

  (d) What is the name of the second entity in this relationship?

  (e) What is the type of the second entity in this relationship?

  (f) What are the keywords should be used to describe the relationship between these two entities?

Similarly, all the information will be stored into a local file and used in our translation algorithm.

2. **SELECT-FROM-WHERE**

   Our search queries have exactly the same syntax and concepts as SQL queries. Users simply need to define all the tables they need using CREATE TABLES beforehand.

   Currently, however, we still do not support aggregation, set operations nor sub-queries.

3.2 Translating SQL into CQL

In this section, we discuss how we can translate a given SQL query into a CQL query.

3.2.1 SQL Queries vs. CQL Queries

SQL queries are table-oriented while CQL queries are entity-oriented. In SQL queries, all the constraints in WHERE statements put constrains on the values of attributes - how the values of attributes relate to each other and whether there are any requirements for attribute values (for example, two attributes should have the same value or a person name should be equal to "Bob"). In CQL, all the constraints in WHERE statements describe the context of entities, which are already implied in the schema of relational tables in SQL queries.

So when given a SQL query, we will use both the constraints in the WHERE statement and the context information the user provided when creating tables to construct constraints describing each entity.

3.2.2 Translation Algorithms

Now we introduce the translation algorithm which translates a SQL query into a CQL query.

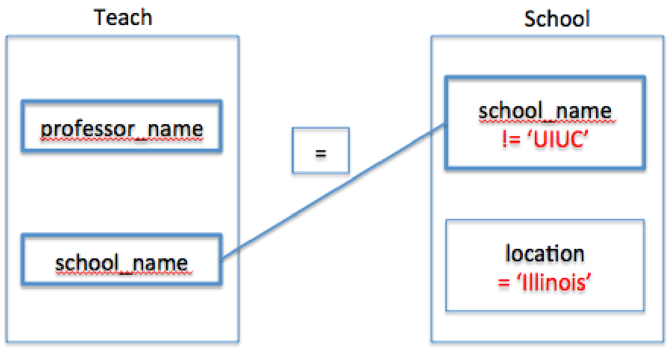


Figure 3.1: A Graph Representation of the given SQL Query

We use the following SQL query as an example. The general algorithm will be presented in section 3.2.4, where the attributes and semantic meanings of tables Teach and School are described in Section 3.1.1.

```
SELECT Teach.professor_name
FROM Teach, School
WHERE Teach.school_name = School.name
AND School.location = 'Illinois'
AND School.name != 'UIUC'
```

To make the illustration easier, we use a graph representation to show all the table definitions and constraints used in the query together. The graph representation of our SQL query is shown in Figure 3.1.

The way to construct the graph is listed as follows:

1. For each table used in the SQL query, we draw a rectangle for it.

2. For each drawn table, we add all the related attributes into it.

   We first add all the attributes directly mentioned in the query into the table. Then as discussed in section 3.1.1, if an attribute is not the only key in the table, it depends on the other keys in the table in order for us to decide whether it satisfies the semantic meaning of the table or not. So we also need to add all the key attributes into the graph.

3. For each constraint in a WHERE statement, if it shows the relationship between an attribute and a value, or the relationships between two at-

tributes inside the same table (for example, "School.location = 'Illinois' "), we will add the constraint inside the rectangle for each table.

4. For each constraint in a WHERE statement, if it shows the relationship between attributes in two different tables (for example, "School.name = Teach.school_name") we can draw a link between the two rectangles and label the relationship on the link.

When translating, we need to make sure that all the information describing the context and all the others constraints are put into the CQL query.

The CQL query contains SELECT, FROM, WHERE and GROUPBY statements. We will start to develop the algorithm from the FROM statement because it defines all the symbols of the entities we are going to use in our query and these symbols will be used in all other clauses.

The FROM statement lists all the different entities that we need to consider in this query - which in our example are Teach.professor_name (PERSON), Teach.school_name (ORGANIZATION), School.school_name (ORGANIZATION) and School.location (LOCATION). A special case we need to consider is when there are attributes linked by '=' operators. These entities are required to have the same value. So one intuitive method to ensure this constraint is to represent all the attributes that are linked together by the '=' operator with the same entity symbol in the CQL query. So in our example, instead of searching for two different ORGANIZATIONs, we only search for one.

In case we are looking for more than two entities of the same basic entity type, we also add an alias for each entity so as to distinguish them. In our algorithm, we format the alias of every entity by simply combining its basic entity type with a unique index.

The final FROM statement we generate for the given SQL query is,

```
FROM #person AS #person1, #organization AS #organization1, #
    location AS #location1
```

The WHERE statement contains two types of patterns: patterns to describe the context of each entity and patterns to describe other constraints.

We first add the patterns describing the context.

In our example, the Teach.professor_name is a PROFESSOR entity. So we need to make sure the PERSON entity we are looking for also co-occurs with the keyword "professor". Thus, we need a pattern "pattern(#person1 professor)" in the WHERE statement. Similarly, we want a pattern for each other specialized entity. One special case is when an entity has a "=" constraint on it, like "location = 'Illinois' " in our example. The value of the entity is already fixed and we do not need to search for it any more. So we can simply replace that entity with the required value every time we need to use it and do not need to add a pattern to describe what type of specialized entity it is.

After this step, the WHERE statement will become,

```
WHERE pattern(#person1 professor) AND pattern(#organization1
   school)
```

For each attribute that is not the only key attribute, we also need to add the patterns showing how it is related to the key entity. For example, in the Teach table, we have two composite keys. In order for a tuple of (professor_name, school_name) to appear in this table, the values for these two attributes also need to co-occur together with the keyword "teach." So we need an extra such as "(#person1 #organization1 teach)".

We will do this for every table. Now the WHERE statement becomes,

```
WHERE pattern(#person1 professor) AND pattern(#organization1
   school) AND pattern(#person1 #organization1 teach) AND
   pattern(#organization1 Illinois)
```

Notice that the pattern "pattern(#organization1 Illinois)" represents the constraint that we want the School.location attribute, which is a LOCATION entity, to co-occur with the school_name attribute, which is an ORGANIZATION entity. But here the ORGANIZATION entity must have the value "Illinois", so we simply replace the entity symbol with the required value. So instead of "pattern(#organization1 #location1)", we use "pattern(#organization1 Illinois)".

At this point, we have added all the constraints related to entity context into the WHERE statement. Now we need to add all the attribute value constraints, which include the constraints on the value of a single entity, for example "school_name != 'UIUC' ", and the constraints on the relationship between two entities, for example "Teach.school_name = School.school_name", into the WHERE statement.

We have already considered the "=" relationship between two entities by using the same hash tag symbol to represent them. We have also considered the constraints that require an attribute have to "=" a value by replacing all the occurrences of that entity in the query by the required value. For all other types of constraints, even though CQL only supports the "=" operator, we are allowed to create user defined functions to represent the meanings in the CQL system. For example, given the constraint "School.school_name != 'UIUC' ", we can create a user defined function called "isDifferent" in the CQL system. The function will compare the values of the two input parameters and return true if the given two entities are different, otherwise it will return false. The pattern showing this constraint is "isDifferent(#organization1, UIUC)".

Our final WHERE statement is

```
WHERE pattern(#person1 professor) AND pattern(#organization1
   school) AND pattern(#person1 #organization1 teach) AND
   pattern(#organization1 Illinois) AND isDifferent(#
   organization1, UIUC)
```

For the SELECT statement, we just need to look back to the SQL query to see what attributes are selected and what entity symbols they map to. In our example, we only need to select the Teach.professor_name attribute, which is represented using #person1.

Thus, the final SELCT statement is

```
SELECT #person1
```

For the GROUPBY statement, we need it because we want the users to see each tuple for only one time. Without GROUPBY, the output of the CQL system shows each occurrence only once. Using GROUPBY can also make sure that the tuples are aggregated before sorting. This can be implemented by using GROUPBY on every entity listed in the SELECT statement.

The final GROUPBY statement we get for our example is

```
GROUPBY #person1
```

3.2.3 Proof of Correctness

Since we can not directly search SQL queries on the same unstructured corpus as CQL queries, when we are proving a SQL query is equivalent to a CQL query, we are actually trying to prove that, if we let a human being to manually build a relational database from this corpus based on the semantic meaning of the domain of interest first, the set of tuples we get by running the SQL query should be exactly the same as the results of running the CQL query on the corpus. Here we only compare the values in the set and ignore the ranking for now. We will talk about the ranking issue in the next chapter.

In the following proof, we assume that we already have such a conceptual database constructed by a human based on the semantic meanings of the domain. Our target is to prove that the result of any SQL query on this database is exactly the same as the result of the CQL query translated from it using our algorithm on the unstructured corpus.

We first make an assumption on the context information the user provides.

Assumption 1 *For the key attribute in each conceptual table, the set of all its values in this table equals to the set of all the possible values that satisfy the user's constraints on the key attribute in the corpus.*

For example, if a user defines a PROFESSOR table, which contains three attributes - PROFESSOR, SCHOOL, DEPARTMENT, he also requires that the key attribute PROFESSOR should be a PERSON entity that co-occurs with the keyword "professor." Then according to Assumption 1, the set of all the PROFESSOR names in this conceptual table equals to the set of all the possible values in the corpus that is a PERSON entity and co-occurs with the keyword "professor."

We first prove that, with this assumption, simple SQL queries are equivalent to the CQL queries translated from our algorithms.

If we are given a query that selects only the key attribute from a table,

Listing 3.1: Case 1

```
SELECT T1.A(K1)
FROM T1
```

the CQL query we will get from it will be of the format

```
SELECT #E1.val
FROM #entitytype_t1 AS #E1
WHERE pattern=("[#E1 keywords]<range>")
GROUPBY #E1
```

The pattern in the CQL query is the user's context requirement on the key attribute.

Since the result of the SQL query is just the set of all the key attribute values in this table, by using Assumption 1, we can know that the result of this CQL query equals to the result of the SQL query on the conceptual database.

Similarly we make another assumption.

Assumption 2 *In any conceptual table, for each pair of key attribute and another attribute in the table, the set of distinct values of this attribute pair in this table equals to the set of all the possible values that satisfy the user's constraints on these two attributes in the corpus.*

In the same PROFESSOR table example, if the user also requires SCHOOL to be an ORGANIZATION that co-occurs with the keyword "university," SCHOOL should co-occur with a PROFESSOR entity and the keyword "work." Then according to Assumption 2, the set of distinct PROFESSOR-SCHOOL value pairs in the table equals to the set of entity tuples that satisfy all the above context requirements in the corpus.

Now if we are given a query that selects one key attribute and another attribute in the same table,

Listing 3.2: Case 2

```
SELECT T1.A(K1), T1.A(K2)
FROM T1
```

the CQL query we will get from it will be of the format

```
SELECT #E1.val, #E2.val
FROM #entitytype_t1 AS #E1, #entitytype_t2 AS #E2
WHERE pattern=("[#E1 keywords]<range>") AND
        pattern=("[#E1 #E2 keywords]<range>") AND
        pattern=("[#E2 keywords]<range>")
GROUPBY #E1, #E2
```

The patterns in the CQL query are all the patterns related to the two required attributes.

Similarly, simply using Assumption 2, we know that the result of this CQL query equals to the result of the SQL query on the conceptual database.

Another special case is when we only select one non-key attribute from the table. As described in section 3.1.1, each non-key attribute value must satisfy its relationship with the key attributes of that table in order to appear in it. Here the result of the SQL query is simply all the distinct values of the non-key attribute in the table. The CQL query translated from it will still have all the context constraints related to both of the attributes, but it only selects the values of the non-key attribute as the final result. The result of each query is the projection of its result of the Case 2 query onto the non-key attribute. Since for Case 2 the results of the two types of queries are the same, their projections onto the non-key attribute are also the same.

Now we consider a more complicated case.

If we are given a query that selects multiple attributes from one table,

Listing 3.3: Case 3

```
SELECT T1.A(K1), ... T1. A(Km1)
FROM T1
```

the CQL query we will get from it will be of the format

```
SELECT #E1.val, #E2.val ...
FROM #entitytype_t1 AS #E1, #entitytype_t2 AS #E2, #
   entitytype_ti AS #Ei ...
WHERE pattern=("[#E1 keywords]<range>") AND
        pattern=("[#E1 #Ei keywords]<range>") AND ...
GROUPBY #E1, #E2, ...
```

Recall that we assume every non-key attribute is functionally determined by the key attribute, and the dependencies involving the key attribute are the only dependencies that we are considering in the system. Thus, if we divide the SQL query into multiple subqueries where each subquery only selects the key attribute and one of the other required attributes, the Cartesian product of the results of these subqueries is the same as directly searching the whole SQL query. On the other hand, in the translated CQL query, we only have the patterns related to each single attribute and the patterns describing the relationships between the key attribute and other attributes. This result can also be achieved by joining the results of the CQL translated from each SQL subquery that selects only the key attribute and another required attribute.

This is because the results of any SQL query and the CQL query translated from it in Case 2 are the same. The Cartesian product of these results are also the same. So in Case 3, the SQL query and the CQL query are also equivalent.

We then try to prove equivalence when the SQL query selects multiple attributes from multiple tables without any extra constraints.

Listing 3.4: Case 4

```
SELECT T1.A(K1), ..., T1. A(Km1), ...... , Tn.A(Kn), ..., TN.A(
   Kmn)
FROM T1, T2, ... Tn
```

Similarly, since there is no dependency between attributes in different tables, the result of a SQL query in Case 4 equals to the Cartesian product of the results of multiple SQL queries in Case 3, where each SQL query only selects from one table. For the translated CQL query, it also does not have any constraints that are related to attributes in two different tables at the same time. So the result is the same as when we find all the tuples that satisfy the constraints of each table and then join them together.

Since the results of Case 3 are equivalent, their joint results are also equivalent.

The last type of query we examine is when the SQL query selects multiple attributes from multiple tables with extra constraints.

This is the most general case:

Listing 3.5: Case 5

```
SELECT T1.A(K1), ... T1. A(Km1), ...... , Tn.A(Kn), ..., TN.A(
   Kmn)
FROM T1, T2, ... Tn
WHERE Ti.Aj = Tp.Aq AND .... AND
    Ta.Ab != Tc.Td AND .... AND
        Tg.Ah = 'keyword' AND ... AND
        Tx.Ay != 'keyword' AND ...
```

Compared to Case 4, Case 5 has some extra constraints added to the SQL query. These constraints will be translated into extra patterns in the CQL query. Both of the constraints and the patterns have the same functionality -

removing tuples from the result of Case 4 if they do not satisfy the new constraints or the patterns. In our translation algorithm, each extra constraint in the SQL query is directly translated into a user defined function which has exactly the same semantic meaning as the constraint. Thus the tuples each constraint and the corresponding pattern in the CQL query that can be removed from the result of Case 4 are the same. So the results for any SQL query and the CQL query translated from it are still the same.

3.2.4 Formal Algorithms

We can formally define the algorithm as:

Algorithm 1 Translation Algorithms

```
1: Input: a SQL query, table definitions
2: For each table occurrence in FROM, store its name in the query and table type into a map $M_{table}$.
3: For each attribute mentioned in SELECT and WHERE statements and the key attributes for each table, group them based on the 'equals to' linking conditions.
4: For each group, give it a new alias #entitytype_index (here the index is used to distinguish different groups of the same entity type), and store the map from this new alias to the set of linked attributes as $M_{group}$. If any of the attributes in this group are assigned values, put those values into another map $M_{groupvalue}$.
5: For all the attributes in the SELECT statement, replace their current names with either the new aliases or the assigned values if they exist and form the new SELECT statement.
6: Create a GROUPBY statement, where we simply replace the "SELECT" keyword in the SELECT statement by "GROUPBY" and do not change the rest.
7: For all the unassigned groups, add the alias into the FROM statement.
8: For each table, if it is a specialized entity table, first create a pattern for its key attribute as "[#entitytype_index context]<range>", where #entitytype_index is the alias of the key attribute. Then for each non-key attribute, create a pattern such as "[#entitytype_index1 #entitytype_index2 context]<range>", where #entitytype_index1 is the alias of the key attribute and #entitytype_index2 is the alias of the non-key attribute. If it is a relationship table, create one descriptive pattern for each of the keys, then create a relationship pattern for both of the keys. If a certain attribute has an assigned value, replace the alias with the assigned value.
9: For each of the constraints that are not "equals to", we convert it into a function and use it as a pattern.
10: Assemble all the patterns into a WHERE statement.
11: Output: Assemble the SELECT, FROM, WHERE and GROUPBY clauses together.
```

CHAPTER 4

RANKING MODEL

4.1 Why Ranking

In our entity search system, users will type a query in the format of SQL query, and the system will return a list of tuples that satisfy that SQL query.

If we are doing this on top of a relational database, there is no need to do any ranking unless the users want to sort the results based on the value of some certain columns. But when doing the search on top of free text webpages, a ranking is naturally needed becuase there is a difference between how likely the found tuples satisfy the given SQL query. The uncertainty is caused by many different factors which we will discuss in the next section.

Thus, in order to sort the results, our target is to calculate, given a tuple u, how likely it satisfies the SQL query q, $P(q|u)$.

4.2 Theoretical Model

The process of generating final results can be divided into three steps:

1. Find all the possible subtuples that satisfy the schema and the constraints inside each table.
2. Join the subtuples based on the join conditions and filter out those tuples that do not satisfy the constraints across the tables.
3. Do a projection and only keep those attributes required by the user.

After the second step, we will get a tuple that is a superset of the final tuple u. This tuple still contains all the information about what subtuples the final tuple is constructed from. We call this tuple a "path" of the final tuple u and denote it using p. We call it a "path" because it tells us through which subtuples we can receive our final tuple. We distinguish different paths by using a subscript p_i.

Now we try to analyze how to factorize our target $P(q|u)$, which is how likely the given final tuple u will satisfy the input SQL query q.

$$
\begin{aligned}
P(q|u) &= \sum_i P(q, p_i|u) && (4.1)\\
&= \sum_i P(q|p_i, u)P(p_i|u) && (4.2)\\
&= \sum_i P(q|p_i)P(p_i|u) && (4.3)
\end{aligned}
$$

We have equation [4.1] based on the definition. Every final tuple u is a projection from its path. The probability that u satisfies the input query q can be divided into the summation of the probability that it satisfies q and it is the projection of one of its paths p_i.

From equation [4.1] to [4.2], we only use the Bayesian rules. Equation [4.2] tells us that the final probability that u satisfies q is influenced by both how likely u is a projection of p_i and how likely u will satisfy query q when it is generated by p_i.

To derive equation [4.3] from [4.2], since all the information contained in u can also be found in p_i, the condition on p_i and u can be simplified into a single condition on p_i.

As we mentioned before, in order to see whether a path satisfies the given SQL query, we need to both check how likely each subtuple in the path satisfies the given query and also how convincing the join is.

One way to model the quality of a join is by checking the context keywords of each pair of subtuples.

For example, given a SQL query,

```
SELECT professor_name
FROM Teach, School
WHERE Teach.school_name = School.name
AND School.location = 'Urbana'
```

The process of generating the final professor names can be modeled as finding all the (professor, school) tuples for the Teach table and all the (school, location) tuples where the location is "Urbana" for the School table. Then we will join the results for the two tables together on the school_name attribute. Finally we will do a projection and only output the professor names. Suppose we find a tuple (Kevin, UIUC) satisfying the TEACH table that appears twice in the corpus. For the first occurrence, we also see the extra keyword "school" appearing closely to the two entities. For the second occurrence, we see the keyword "visit" appearing next to them. Suppose for the SCHOOL table we find a tuple (UIUC, Urbana) that satisfies the semantic requirements. It only has one occurrence in the corpus and appears close to the keyword "school." It is more reasonable to believe we should join the first occurrence of (Kevin, UIUC) to (UIUC, Urbana) than to the second occurrence. Because while the tuple contents are the same, the context keywords of the first occurrence are more related to the context of the tuple to be joined with.

But the problem of considering the context of each subtuple when evaluating the confidence of a joined tuple is that its memory and time complexities are both very high. This is because in order to compare how closely the contexts of two tuples are related to each other, we need to maintain the extra information about all the keywords that have appeared around each tuple occurrence. And when evaluating the confidence of a joined tuple, we need to evaluate the confidence of the product of each occurrence of each subtuple and sum them together. Since the number of tuple occurrences is much larger than the number of distinct tuples, the increase in complexity is significant.

In order to make the complexity fall within an acceptable range, we assume that each subtuple is independent from each other. This means the probability of a path satisfying the input query only depends on how likely each of the subtuples in the path satisfies the corresponding table, and it does not rely on how likely the two tuples can be joined together.

Based on this independence assumption, we can further derive the above equation. In the following derivation, we use t_j to represent each table used in the SQL query. As we described before, each path is generated from a set of subtuples that each satisfies the semantic requirements of the table. The subtuple that generats p_i and satisfies T_j is represented using p_{ij}. We use $count(p_i)$ to represent the total number of times that tuple p_i has appeared in the whole corpus. And we denote $\sum_i count(p_i)$ as $count(p)$.

Since the probability of each subtuple satisfying the corresponding table is independent to the corresponding probabilities of other tuples, we can divide $P(q|p_i)$ into the product of $\Pi_j P(t_j|p_{ij})$, which gives us equation [4.4] as follows:

$$P(q|u) = \sum_i \Pi_j P(t_j|p_{ij}) P(p_i|u) \tag{4.4}$$

$$= \sum_i \frac{count(p_i)}{count(p)} \Pi_j P(t_j|p_{ij}) \tag{4.5}$$

From [4.4] to [4.5], we further derive the probability that the final tuple is generated from path p_i. Since u can be the projection of many different paths, the probability that it is from p_i is in proportion to how many times p_i has appeared in the corpus. Thus we use $\frac{count(p_i)}{count(p)}$ to implement $P(p_i|u)$, where $count(p)$ is the normalization factor.

4.3 Implementation

Given our final derivation of $P(q|u)$ in [4.5], we can calculate $count(p_i)$ and $count(p)$ by counting how many times the tuples appeared in the corpus. In order to calculate the result of $P(q|u)$, we just need to define how to calculate $P(t_j|p_{ij})$.

Since $P(t_j|p_{ij})$ represents how likely a subtuple p_{ij} matches the given table t_j, we list the following criteria that we think are necessary when evaluating how likely they will be matched:

1. How likely does each entity value inside the tuple satisfy the required target type?

2. For each non-key attribute, how likely does it have the specified relationship in the schema?

3. How likely does the given tuple satisfy each of the constraints?

We can see that for each of the criteria we listed above, they are translated into one pattern (in the format of one entity search query) in the CQL query. Here we do not consider how the score for each pattern is calculated because the EntityRank system already considers sufficient factors. So we only need to know when, given the score for each pattern, how can we aggregate them and generate the final $P(t_j|p_{ij})$.

We use s_{ij_k} to denote the score of a pattern pt_{ij_k} of subtuple p_{ij}. We also assume that each pattern influences whether a subtuple will satisfy the table requirements with equal importance independently. Then we can get the probability that the subtuple satisfies the given table by calculating the product of the scores for all the patterns without using any weight.

Finally we have $P(t_j|p_{ij}) = \Pi_k s_{ij_k}$.

CHAPTER 5

SYSTEM DEMO AND EXPERIMENTAL RESULTS

5.1 Experimental Settings

In this system, we use Wikipedia as our underlying web corpus. We use the StanfordNER tool to tag “PERSON,” “LOCATION” and “ORGANIZATION” entities and to use them as our basic entities.

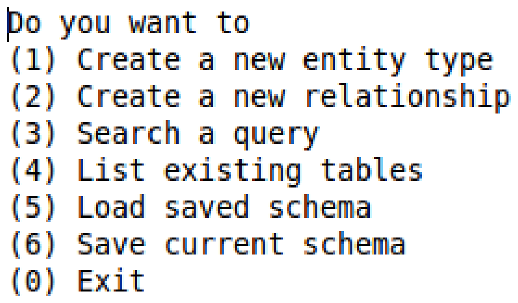
We designed three set of queries. We use the first query to illustrate how to use the demo system. For the other two types of queries, we test our system by assigning different values to the attributes and calculating and analyzing the precision@10 of the results. We do not evaluate the recall here because the cost of counting how many valid tuples there are in the whole corpus is too expensive.

5.2 Case Study and Demo Interface

5.2.1 Interface Design

Our system provides the basic function for users to create and manage their own tables and to search via SQL queries.

We list all the functions we support for now in Figure 5.1. Users can create new tables (either a new compound entity type or a new relationship), view all the existing tables, load or save schemas and search via SQL queries using existing schemas.



```
Do you want to
(1) Create a new entity type
(2) Create a new relationship
(3) Search a query
(4) List existing tables
(5) Load saved schema
(6) Save current schema
(0) Exit
```

Figure 5.1: Function List

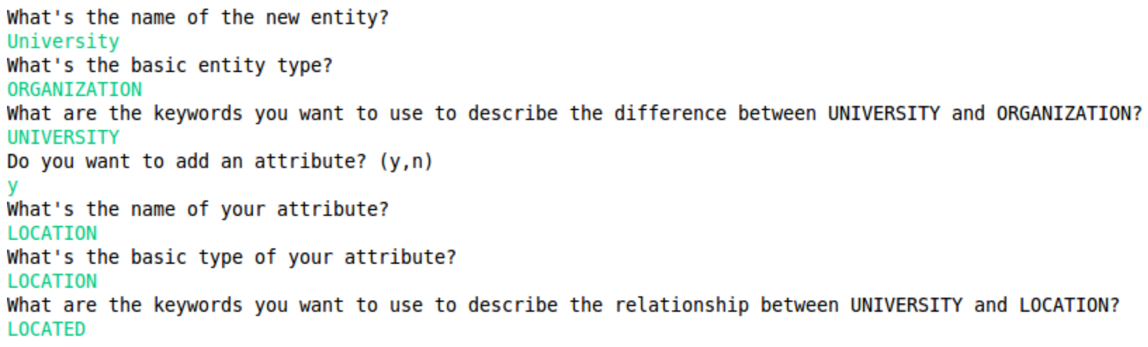
5.2.2 Translation Results

In this example, we go through the whole querying process using an example where the user wants to find all the professors that work in a particular university.

We can get the results following these steps:

1. **Create a new entity type called "University"**

   By following the instructions listed in Figure 5.2, the user can create a "University" entity type, which is an organization that co-occurs with the keyword "university." It also has an attribute called "location," which is related to the university through the "located" keyword.



```
What's the name of the new entity?
University
What's the basic entity type?
ORGANIZATION
What are the keywords you want to use to describe the difference between UNIVERSITY and ORGANIZATION?
UNIVERSITY
Do you want to add an attribute? (y,n)
y
What's the name of your attribute?
LOCATION
What's the basic type of your attribute?
LOCATION
What are the keywords you want to use to describe the relationship between UNIVERSITY and LOCATION?
LOCATED
```

Figure 5.2: Add University Entity

2. **Create a new entity type called "Professor"**

   Similarly, we can define a "Professor" entity type, which is a person that co-occurs with the keyword "professor."

3. **Create a new relationship called "Teach"**

We create a “Teach” relationship type, which is a relationship between a “Professor” and a “University.” And the two entities should co-occur with the keyword “teach” or “work.”

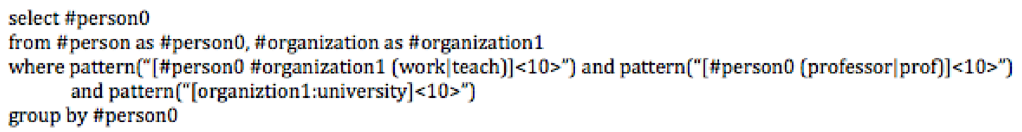
4. **Search via SQL Queries**

   After defining all the new entity types and relationships, the user can use the following SQL query to look for all the university entities that are located in Illinois:

```
SELECT teach.professor
FROM teach
```

   If the input SQL query is valid, the system will translate it into a CQL query.

   The CQL query translated from the given SQL is shown in Figure 5.3.



```
select #person0
from #person as #person0, #organization as #organization1
where pattern("[#person0 #organization1 (work|teach)]<10>") and pattern("[#person0 (professor|prof)]<10>")
        and pattern("[organiztion1:university]<10>")
group by #person0
```

Figure 5.3: CQL Query

   We can see that a lot of context information (for example, this professor needs to work at a university) are automatically added into our CQL query.

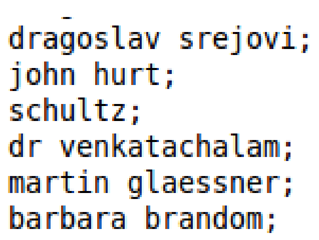
   This CQL query will then be passed to the CQL search system.

   Finally the system will return a list of tuples and sort them based on the probability that they satisfy the given SQL query.

   The top returned results are listed in Figure 5.4.

5.3 Experimental Results

To experiment, we test two different sets of queries. Before querying, we first define an entity type “university,” an entity type “professor” and a re-



```
dragoslav srejovi;
john hurt;
schultz;
dr venkatachalam;
martin glaessner;
barbara brandom;
```

Figure 5.4: Top Results

lationship “teach.” A “university” entity is defined as an organization that has co-occurred with the keyword “university.” It can have the attribute “location”, a “location” entity, which co-occurs with “university” and the keyword “locate.” A “professor” entity is a person who has the keyword “professor” appearing near his name. The “teach” relationship contains two entities, “professor” and “university,” which should co-occur with the keywords “work” or “teach.”

5.3.1 Search for Universities

The first set of queries we experimented with is

```
SELECT university.name
FROM university
WHERE univeristy.location = $state
```

In this set of queries, we are interested in all the universities that are located in the given state.

We randomly sampled 10 different states and calculated the precision@10 of our system by manually checking whether the result is a university located in the required state. The average precision for these ten states was 81%.

We further examined the results to determine the main reasons causing these errors.

- The main reason is that our constraints used to filter whether or not the given organization is a university are not perfect. Some organizations like “usc university hospital” or “museum of texas tech university” contain keywords “univeristy” and are in the required state, but they are just organizations related to universities. Among the 19 errors, 12 of them are caused by this reason.

- Another source of errors was the ambiguity of a certain value. For example, we require the school to be located in a certain state, but the state name can also be the value of some other attribute. For example, the tuple "University of Minnesota" appears in the results for schools in Washington state, because it is located on Washington Avenue. We saw 3 ambiguity errors in our experiments.
- Another source of errors is the tagging error. For example, in our results "university avenue and walnut stree" is tagged as an "ORGANIZATION" but it is actually a "LOCATION." Among the 19 errors we encountered, 2 of them were mistagged as "ORGANIZATION" entities. In some other cases, the HTML file was not parsed very well and some HTML tags were mixed into the content and wrongly tagged by the NER tool as an entity. This caused 2 extra errors.

5.3.2 Search for Professors and Univerisities

The second set of queries we experimented with is

```
SELECT Teach.University, Teach.professor
FROM Teach, University
WHERE Teach.University = University.name
    AND University.Location = $state
```

In this set of queries, we are interested in all the (professor, university) tuples, where the universities should be located in a given state.

We also randomly sampled 10 different states and calculated the precision@10 for each state. But for all the 10 different states we could only find 5 tuples, even though the resulting precision was 100%.

We think this low recall is caused by the fact that there are too many constraints in the queries. So we loosened the constraints by removing the requirement that the "location" attribute has to co-occur with the keyword "locate." After that, the average number of tuples we found for each state reached 20 with the average precision at 85%.

When analyzing the source of the errors, we found that the main source was mistagging. For example, the tagging tool consistently thought

"Ann Arbor" is a person name and ranked the tuple "(University of Michigan, Ann Arbor)" highly. This mistagging alone caused 13 of the total errors we encountered. By comparison, imperfect constraint rules caused significantly fewer errors. We think since we are using multiple constraints in our query, each constraint provides a small improvement in accuracy. Thus, the final results have very few errors that are caused by constraints.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this work, we designed a structured querying system for unstructured webpage data. The system combines the advantages of the expressiveness of structured queries and the flexibility and low maintenance that comes with an unstructured dataset. We also provide a ranking model, which explains the necessities and principles of ranking the result tuples. We implemented our algorithms into a translation system built upon the CQL system. This system allows users to define their own tables and write SQL queries using the tables they defined to search on an unstructured corpus.

There are still multiple directions that we would like to explore.

1. **Support for More Complex Queries**

   Currently we only support basic SELECT-FROM-WHERE statements. But extensions can be done to support aggregation, set operator and nested SQL queries.

2. **Optimization of Search Efficiency**

   Currently only simple optimization based on index selection is done inside the CQL system. That is, we do not have any optimization in terms of how to generate a more efficient CQL query when given a SQL query.

3. **Direct Translation of SQL Queries Into Entity Search Queries (Bypassing the CQL Query Layer)**

   Currently we translate SQL queries into CQL queries first, and then directly call the entity search system using the CQL system. We did this because our focus is on the conceptual design of connecting a structured query with unstructured webpages. But this intermediate step is not necessary. Skipping this layer may help with optimization.

4. **Support for Entity Resolution**

   Currently our system makes the assumption that if two entities have the same name, the they are the same real world entity. Thus, including entity resolution into the problem may help us to increase the precision of the results.

REFERENCES

[1] T. Cheng, X. Yan, and K. C.-C. Chang, “Entityrank: searching entities directly and holistically,” in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 387–398.

[2] T. Cheng and K. C.-C. Chang, “Entity search engine: Towards agile best-effort information integration over the web.” Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2007.

[3] G. K. F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum, “Naga: Searching and ranking knowledge.”

[4] M. J. Cafarella, C. Re, D. Suciu, O. Etzioni, and M. Banko, “Structured querying of web text,” in *3rd Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar, California, USA*, 2007.

[5] S. Chaudhuri, R. Ramakrishnan, and G. Weikum, “Integrating db and ir technologies: What is the sound of one hand clapping?” in *CIDR*, vol. 5, 2005, pp. 1–12.

[6] F. M. Suchanek, G. Kasneci, and G. Weikum, “Yago: A large ontology from wikipedia and wordnet,” *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 6, no. 3, pp. 203–217, 2008.

[7] B. Tan and F. Peng, “Unsupervised query segmentation using generative language models and wikipedia,” in *Proceedings of the 17th international conference on World Wide Web*. ACM, 2008, pp. 347–356.

[8] K. M. Risvik, T. Mikolajewski, and P. Boros, “Query segmentation for web search.” in *WWW (Posters)*, 2003.

[9] X. Yu and H. Shi, “Query segmentation using conditional random fields,” in *Proceedings of the First International Workshop on Keyword Search on Structured Data*. ACM, 2009, pp. 21–26.

[10] C. Zhang, N. Sun, X. Hu, T. Huang, and T.-S. Chua, “Query segmentation based on eigenspace similarity,” in *Proceedings of the ACL-IJCNLP 2009 Conference Short Papers*. Association for Computational Linguistics, 2009, pp. 185–188.

[11] M. Zhou, T. Cheng, and K. C.-C. Chang, “Docqs: a prototype system for supporting data-oriented content query,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 1211–1214.

[12] M. Zhou, T. Cheng, and K. C.-C. Chang, “Data-oriented content query system: searching for data into text on the web,” in *Proceedings of the third ACM international conference on Web search and data mining*. ACM, 2010, pp. 121–130.