In compliance with the
Canadian Privacy Legislation
some supporting forms
may have been removed from
this dissertation.

While these forms may be included
in the document page count,
their removal does not represent
any loss of content from the dissertation.

Towards an Intelligent Information System

By

Xiaofei Jia

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

May 28, 2003

© Copyright

2003, Xiaofei Jia

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitions et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-88972-6

Our file Notre référence

ISBN: 0-612-88972-6

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Canada

Abstract

Nowadays, information overload is becoming an ever-growing problem. As a result, finding the exact information is quite difficult. One major reason is that the information is not organized according to its natural structures. There is not a data model that can represent unstructured or semi-structure data in a natural way. Moreover, almost every information application exploits interactive interfaces that are still ad-hoc. Therefore, how users express their queries in a semantic way is an important research topic. This thesis presents the design of an effective conceptual model to organize information according to its natural structures, based on object-relation databases, and describes a quasi-natural language interface for the conceptual model. It discusses the design and implementation of our prototype system.

Acknowledge

First of all I would like to express my sincere thanks to my supervisor and mentor Dr. Mengchi Liu, for his invaluable help, advice, and financial support since starting my studies at Carleton University. I also appreciate the TA offer from the School of Computer Science.

I would like to express my gratitude to my wife, Shenge Ding, for her support and understanding. I am grateful to my parents for their encouragement and support.

I would like to thank my fellow lab mates Wenjie Tong, Lingyan Xu, Tao Fu and all other friends who give me help during my graduate studies.

Contents

Abstract	iii
Acknowledge	iv
List of Tables	viii
List of Figures	x
1 Introduction	1
1.1 Motivation	1
1.2 Goals of this Thesis	3
1.3 Thesis Contributions	4
1.4 Thesis Organization	5
2 Background	6
2.1 Data Model	6
2.2 Object-relational Data Model	8
2.2.1 Overview	8
2.2.2 Type Extension	10
2.2.3 Nested Relational Data Model	14
2.2.4 Type Inheritance	14

2.2.5	Object Tables, Row Objects, and Column Objects	15
2.2.6	Object Identifier	16
2.2.7	Object Navigation	17
2.2.8	Chapter Summary	18
3	Related Work	19
3.1	ER and EER Conceptual Data Model	19
3.2	Natural Language Interfaces to Databases	20
4	Category Model	23
4.1	Objects and Attributes	24
4.2	Categories	26
4.3	Diagram Representation	27
4.4	Category Hierarchy	29
4.5	Relationships in the Category Model	30
4.5.1	Dependency Relationships	33
4.6	Example	34
4.7	Database Design	37
4.7.1	Object-relational Database Design using the Category Model .	37
4.7.2	Mapping Examples	39
4.8	Features of the Category Model	41
5	Natural Language Patterns	43
5.1	Concepts	44
5.2	Patterns	45
5.2.1	Pattern 1	45
5.2.2	Aggregation	48
5.3	Advantages and Limitations	50

5.4	Chapter Summary	51
6	Implementation of the Prototype System	52
6.1	Abstract Models	52
6.2	System Architecture	54
6.3	Identifying the meanings of Keywords	56
6.3.1	Keyword Search	57
6.3.2	Identify the Categories of Keywords	59
6.4	Alias Matching Manager	61
6.4.1	Alias Matching Using Regular Expressions	61
6.4.2	Alias Matching in Database Level	62
6.5	Parsing	62
6.5.1	Getting Three Elements	63
6.5.2	ORSQL Generation	67
6.6	Some Examples	72
6.7	User Interface	76
7	Conclusion	81
7.1	Future Work	84
	Bibliography	85

List of Tables

4.1	Types Mapping	38
4.2	Tuples with different object types stored in one object table	39
6.1	Reserved words and corresponding relation operators	65
6.2	Queries and ORSQL	77

List of Figures

2.1	Database classification matrix	9
2.2	Retail store schema	11
2.3	Illustrating a nested relation	15
2.4	An example of type inheritance	16
4.1	Category Model Example	25
4.2	Category Person	28
4.3	Category Hierarchy Example	29
4.4	Inheritance in the Category Model	31
4.5	Notation for Multiplicity in the Category Model	32
4.6	Relationships in the Category Model	33
4.7	Existence Dependency Relationship	34
4.8	Data Modeling Example	36
6.1	Abstract models in NLIDB	53
6.2	System Design	54
6.3	System Architecture	55
6.4	Screen Shot of the Keyword Search	57
6.5	Example of different objects with the same name	58
6.6	Query tree for example 6.5.1	64
6.7	Relationships among category, object types, and object table	68

6.8	Results of the query: Find name, email, website of professors in Carleton University whose research includes database.	72
6.9	User interface: select related categories	73
6.10	Results of the query: Find cities in Canada whose population is more than 0.3 million	75
6.11	Results of the query: Find countries whose population is maximal. . .	76
6.12	User interface	78
6.13	Country Canada	79
6.14	Provinces in Canada	80
6.15	Province Ontario	80

Chapter 1

Introduction

1.1 Motivation

Although there are tremendous amount of data at both the enterprise and Internet level, the lack of structure makes finding the right knowledge exceedingly difficult. For example, the current search engines use the index, words matching and ranking techniques to deal with information syntactically rather than model the web documents according to their contents. Therefore, what one usually finds from the current search engines is irrelevant information. How to model and query these unstructured or semi-structure data according to their semantic meaning in a natural way is vital.

Our approach for data modeling is based on the object-relational database (ORDB) due to its significant capabilities for handling complex data. Although ORDBs have been considered “the next great wave” [6] in the database research community, the application, attempting to solve the mentioned problems, is still difficult to develop, as there is not a well-defined conceptual model for such kind of database design that plays the same role as ER or EER model [10] for the relational model. As a result,

many users have a great deal of trouble with database design in ORDB, and poor schema design is responsible for the failure of many applications. Defining a conceptual model based on the object-relational concepts not only can reduce the difficulty of the application development on the ORDB but also makes data meaningful.

On the other hand, although database query languages, such as SQL, are very powerful for data access, real database users (end users) do not know how to use them. Almost every information application exploits interactive interfaces in some way. However, the development of user interfaces is still ad-hoc [17] that often use predefined formed-based interfaces. This greatly limits what users can do with the data in databases. How to express queries in a semantic way is an important research topic.

Natural language interfaces to databases (NLIDB) allow users to use natural language to express database queries so that users do not have to have the knowledge about the structure of the database. The successful development of NLIDB will greatly expand database applications. However, the development of natural language interfaces has been a challenge since 1960s. One of main reasons is the portability problem [29, 38], which is caused by the limitation of the knowledge domain representation. In the past, a lot of researches on NLIDB have been done based on the relational model. However, the relational model cannot represent the real world applications in a natural way, because it requires a complex normalization process that results in the data about real world entities to be scattered in many relations. As a result, there are still no real natural language interfaces that can be used for general database applications.

Based on the object-relation database, the major objectives of this thesis include:

(1) designing an effective conceptual model to organize the semi-structure or unstructured data, and (2) building an information system that provides a natural language interface to bridge the gap between users' query needs and the implementation of the system.

It is our hope that the conceptual model and the natural language interface proposed herein would help the ongoing efforts to build a good schema, and hence pave the way for the applications of this new generation database management system.

1.2 Goals of this Thesis

One purpose of the thesis is to define a new conceptual model, called the category model, which maintains the semantic meaning of the data, better integrates object-relational features, and can be easily mapped to the object-relational database schema. As a conceptual model, it can be applied to any applications based on the object-relational database and has significant advantages.

Another important purpose of this thesis is to build a quasi-natural language interface to ORDB based on the category model. Our approach of natural language processing does not rely on grammar. Instead, we fully utilize database semantics to resolve linguistic ambiguities. In particular, we provide simplified natural language patterns for users to construct their queries. Based on these patterns, we build a natural language parser to translate the natural language queries directly to ORSQL rather than transforming queries into an intermediate logical query. In addition, the system portability is ensured by the wide domain coverage of the category model.

1.3 Thesis Contributions

The main contributions of this thesis can be described as follows.

1. Build a new conceptual model-the category model that provides a natural way to organize the complex data with semi-structure or unstructure, as well as a means to easily map the object-relation database schema. In particular, the category model has the ability to:
 - (a) Maintain the data semantic meanings.
 - (b) Combine the best of the relational data model and object-oriented data model.
 - (c) Make the implementation of the natural language processing easy.
2. Demonstrate how to use the category model to model the real world entities and relationships and how to map this conceptual schema to object-relation database schema.
3. Design several natural language patterns. So that end users can use quasi English to express their queries without knowing the database structure.
4. Develop a prototype system. As a proof of the concept, we have developed a prototype system called Intelligent Information System. In this system, we use the category model to create a conceptual schema which contains Canadian provinces and cities, and all countries in general, Canadian universities and their faculties, professors, staff, departments, and programs. The data we gathered is the raw data from the web. The category model classifies the data based on their natural categories, category hierarchies and the relationships among categories. Therefore, the system contains highly structured information rather than data

without meaning. The high structured information allows comparison, calculation and inference based on the contents rather than the syntactically retrieval. Utilizing natural language patterns, users can construct their queries in a semantic way, such as “*Find the professors in Carleton University whose research includes databases*”, or “*Find the countries whose population is more than 30 million and GDP is not less than 2000*”.

1.4 Thesis Organization

The remainder of this thesis is organized as follows:

- Chapter 2 presents some background concepts that are important to understand this thesis.
- Chapter 3 reviews the related work.
- Chapter 4 describes the concepts of the category model and introduces the method for mapping the category model to the object-relational database schema.
- Chapter 5 introduces several natural language patterns.
- Chapter 6 focuses on the implementation of the prototype system.
- Chapter 7 presents the conclusion of this thesis and describes the directions for future research.

Chapter 2

Background

The object-relational database enhances the capabilities of the relational database by incorporating some of the key features in the object-oriented database. In particular, the object-relational database introduces the features of object identifier, object types, methods, type inheritance, and nested relations. This chapter provides background concepts that are important to understand the motivation of the thesis. We start, in Section 2.1, by discussing data models which are fundamental to study the database systems. Particular attention is given to the concepts of object-relational data model in the Section 2.2. Through illustrations of the features of the object-relational database, we explain why we chose object-relational database as our research database system. Section 2.3 summarizes the Chapter.

2.1 Data Model

A database is a permanent, self-descriptive repository of data that is stored in one or more files [22]. In order to describe data with various structures and relationships,

we need provide some level of data abstraction by concealing details of data storage. The data model is a collection of concepts that can be used to describe the structure of a database – it provides the necessary means to achieve data abstraction [37]. It is comprised of data types and structures, operations, and (consistency) constraints. In other words, it is a pattern according to which data are logically organized and manipulated. A data model is concerned with the representation of real world entities as well as their organization and interrelationships.

There are many data models proposed. From the type of concepts they use to describe the database structure, data models can be categorized as *high-level* or *conceptual data models*, *lower level* or *physical data models* and *representational* or *implementation data model* [7, 24, 31, 37, 45].

- **Conceptual data models**

Conceptual data models provide concepts that are close to the way many users perceive data. The **ER model**, and its variant **EER model** are today's most popular conceptual models which use concepts such as entities, attributes, and relationships to describe the real world entities. Object data models can be regarded as a new family of conceptual models [4, 37] which share a number of the basic concepts, such as the concepts of *object*, *class*, and *inheritance*.

- **Physical data models**

Physical data models provide concepts that describe the details of how data is stored in the computer. It is generally meant for computer specialists, not for typical end users.

- **Implementation data models**

Implementation data models provide the concepts that may be understood by end users but that are not too far from the way that data are organized within

the computer. It conceals some details about data storage but can be implemented in a direct way. The implementation data models include the widely used relational data model, as well as the so-called legacy data models – the network and hierarchy models that were widely used in past.

2.2 Object-relational Data Model

The early object-relational database vendors were start-up companies that included Illustra, Omniscience, and UniSQL. In February 1996, Illustra was acquired by Informix, who immediately announce that the Informix relational database would be combined with the Illustra server to create an object-relational engine. More recently, Oracle, IBM, and Sybase released their object-relational database products. In addition, some of the object vendors have been adding SQL engines to their products [6]. Obviously, there is significant latent demand for object-relational capabilities.

2.2.1 Overview

The broad idea of every case mentioned above is that the product should support both object and relational capabilities. Databases that support a dialect of SQL-3, include non-traditional tools, and optimize for complex SQL-2 queries, which are called object-relational databases [6]. They are relational because they support SQL; they are object-oriented because they support complex data and object-oriented features, such as inheritance, polymorphism, encapsulation, etc. A *classification matrix* for databases [6] is shown in Figure 2.1.

In this two-by-two matrix, the horizontal axis illustrates the ability of applications in handling the simple data and complex data. The vertical axis differentiates whether

Figure 2.1: Database classification matrix

the application requires a query capability.

- **Quadrant 1** of the matrix represents applications that deal only with simple data and have no requirement for ad-hoc query. The traditional word processor is a good example.
- **Quadrant 2** represents applications that require an ad-hoc query but still deal only with simple data. The traditional relational databases fall into this quadrant.
- **Quadrant 3** represents applications with complex data and processing requirements but no ad-hoc query requirement. Current object-oriented databases fall in this quadrant. For example object-oriented databases in the CAD/CAM applications.

- **Quadrant 4** represents applications with a need for both complex data and ad-hoc queries for that data. The object-relational databases fall into this quadrant by its support for complex data inherited from object-oriented database and query capability supporting SQL.

2.2.2 Type Extension

The object-relational database enables users to define data types and functions (methods). In relational database, the type of an attribute must be atomic, only the following types can be used in the SQL-92:

- Integer
- Floating-point number
- Character string, fixed or variable length
- Date, time, datetime, interval
- Numeric and decimal

Because SQL's set of data type and operations are limited, many real world problems are difficult to code, and once coded, perform badly. The type extension makes these kind of problems easy to handle. Now, we illustrate an example to explain the importance of the type extension. Suppose that, in a big retail company with many stores distributed world wide, we want to write a program to manage the store's stock. When one sub-store's stock is low, we want to be able to find another store near to this one to help replenish the low quantity of stock. In a SQL-92 system, the schema about the *store* table is shown in Figure 2.2(a). Although every store's mailing address is available in this table, the street address, city, province and zip code

Figure 2.2: Retail store schema

are not the attributes to appropriately describe the concept *near*. The best solution is to record the geographic position of each store as a (latitude, longitude) point. Then we can find neighboring stores which are within any given distance of each other. We can add attributes *lat* and *long* to the table *store*. The schema is shown in Figure 2.2(b), where (lat, long) are the coordinates of a store's address. Suppose we want to find the stores near an *Ottawa branch* within a distance of 20 miles. The SQL is shown as follows

```
select s.name
from store s, store q
where q.name = 'Ottawa branch' and
(s.long - q.long) **2 + (s.lat - q.lat) **2 < 400
```

The problem here is the necessity of simulating the data type *geographic point* in SQL-92 because it is not its supporting type, it must be simulated by using two numbers (lat, long). Moreover an operation is needed to find the points that lie within a designated circle. In addition, because this operation is not in SQL-92, it must be simulated by coding a collection of numeric operations. This simulation is sufficiently

complex and the query optimizer cannot execute the resulting query efficiently.

On the contrary, the extensible data types available in the object-relation database eliminate the awkward type simulations that cause efficiency problems. Here is the better way to solve the problems presented by the example given above. In some object-relational database, such as Oracle, Informix, *Point* is one of a 2-D geometric data type which support 2-D geometric functions. (Of course, we can define an object type like this in the object-relational database ourselves). By using the point data type we can add the attribute *location* to the *store* table which is shown in Figure 2.2(c):

There is also a function called *distance* to calculate the distance between two points. So, the *ORSQL* can be written as follows:

```
select s.name
from store s, store q
where q.name = 'Ottawa branch' and
distance(r.location, s.location) < 20
```

The query is more understandable for users and more efficient due to the technologies supporting type extension, such as *Dynamic linking* [6, 37].

In order to meet the needs for the applications with complex data types, current commercial object-relational database products, such as Oracle 9i and Informix Universal Server, provide built-in new data types needed in multi-media, financial, spatial, and other applications. They also provide a framework for database extensibility so that new multimedia and complex data types can be supported and managed natively in the database. This framework provides the infrastructure needed to allow extensions of the data server by third parties.

Except for the data type extensions, object-relational database allows users to define additional kinds of data which specify both the structure of the data and the ways of operating on it. User-defined data types make it easier for developers to work with complex data types. There are two categories of user-defined data types: Object types and Collection types.

- Object types are abstractions of real world entities which store structured object entity data in its natural form. For this reason, they are suitable for describing entities with complex types in the real world. An object type is a schema object with three kinds of components: a name, attributes, and methods. A name serves to identify the object type uniquely within the schema. Attributes model the structure and state of the real-world entity. Attributes are built-in types or other user-defined types. Methods are functions or procedures that can be written in either ORSQL or in general-purpose third-generation programming languages, such as C and Java, and then registered to a system. This approach integrates the power of programming language into database systems. Methods of an object type model can be considered the behaviours of objects.
- Each collection type describes a data unit made up of an indefinite number of elements, all of the same data type. The collection types are array types and table types. Array types and table types are schema objects. The corresponding data units are called *varrays* and *Nested tables*.
 - A varray is an ordered set of data elements. Each element has an index, which is a number corresponding to the element's position in the array.
 - A nested table is an unordered set of data elements. It has a single built-in type or an object type column, and the type of that column is a built-in type or an object type. If the column is an object type, the table can be

viewed as a multi-column table, with a column for each attribute of the object type. Nested tables can contain other nested tables.

Object-relational database provides users with the ability to model their own objects by enhancing the type system to support user-defined types. These types are meant to closely model application objects and are treated as built-in types, such as number and character, by the database server.

2.2.3 Nested Relational Data Model

As described in the Subsection 2.2.2, the collection type is one of the complex object features. It can be used to represent the nested relational model.

The nested relational model removes the restriction of first normal form (1NF) from the basic relational data model, and thus is also known as the **Non-1NF** or **NF²** relational model. The relational data model is called the **flat relational model** because attributes are required to be single-valued and to have atomic domains. The nested relational model allows for composite and multi-valued attributes, thus, leading to complex tuples with a hierarchical structure. This is useful to represent the objects that are naturally structured hierarchically. Figure 2.3 (a) shows a nested relation table schema for *Department* in a specified university and Figure 2.3(b) provides an example of a nested tuple in table *Department*.

2.2.4 Type Inheritance

An object-relational data model supports the inheritance, the same manner as the object-oriented data model. An object type can be created as a subtype of an existing object type. A subtype inherits all the attributes and methods of its super-type, can

(a)

Name	Director	Professors			Programs	Contact Information
		Name	Students			
			Name	Student Number		

(b)

Computer Science	Eric	John	Bob	21221	Ph.D of Computer science	627 labeo Building. 34 Suasn St. Ottawa, Ontario. K1S 5B6 Canada
			Mary	25881	Master of Computer Science	
		Jim	Tom	29101		
			Lisa	32122	M.Sc. in Information Systems Science	
			Mike	29324		
		Frank	Colin	21348	B.Sc (Honours) Combined Program	
		Paul				

Figure 2.3: Illustrating a nested relation

add new attributes and methods, and can override any of the inherited methods. Figure 2.4 illustrates two subtypes, *student* and *employee*, which inherit from the super-type *person*. Furthermore, a subtype can define another subtype, thus type hierarchies are built up. For example, in Figure 2.4, *Part time student* and *Full time student* are derived from the super-type *Student*.

2.2.5 Object Tables, Row Objects, and Column Objects

An **object table** is a special kind of table that holds objects and provides a relational view of the attributes of those objects. For example, the following statements define an object table *Country_tab* according to object type *Country.t*:

Figure 2.4: An example of type inheritance

```

create or replace type Country_t
    Name varchar(20),
    Population float,
    .....);
create table Country_tab of Country_t
.....

```

Objects that appear in object tables are called **row objects**. Objects that appear in table columns or as attributes of other objects are called **column objects**.

2.2.6 Object Identifier

In the object-relational database, every independent object is assigned to a unique identity called **object identifier** or **OID**, which is generated by the system automatically. Once the OID is generated, it is immutable; that is, the OID value of a particular object can not change. Utilizing this property, we can distinguish one object by its OID, rather than the *primary key* in the relational database. One OID can only be assigned to one object once. Even when an OID is removed from the system,

its OID can not be assigned to another. In other words, the OID doesn't depend on the attributes of objects because the value of the attributes can be modified. This property naturally represents the identity of the real world object. Another distinguished property of the OID is that it can be used as means to reference one object to another. Utilizing this property, we can use the OID to represent the relationships among objects.

The *circular reference* refers that a column of a relation can reference to a row object of an object table. In relational database, the *circular reference* is impossible. For example, if we define a schema in the relational database to represent a person with children as: $person = [name:string, children:\{person\}]$, one of tuples should be: $[name \Rightarrow 'Tom', children \Rightarrow \{[name \Rightarrow 'Bob', children \Rightarrow [...]]\}]$. The values of attribute *children* fall into a loop without an exit. In the object-relational database, on the contrary, we can define a schema for person as: $Person = [name: string, spouse: Person, Children: \{Person\}]$. One tuple may be: $o1 [name \Rightarrow 'Tom', spouse \Rightarrow o2, children: o3, o4]$. $O2, o3, o4$ are OIDs of the tuples: $o2 [name \Rightarrow 'Pam', spouse \Rightarrow o1, children \Rightarrow o3, o4]$, $o3 [name \Rightarrow 'Sam']$, $o4 [name \Rightarrow 'Jan']$. Thus, the *circular reference* can naturally represent this kind of relationships.

2.2.7 Object Navigation

The query language in object-relational database is ORSQL which is a dialect of SQL-3 [6]. The SQL statements have the same structure of SELECT-FROM-WHERE-GROUP BY-HAVING-SORT BY but support inheritance, reference, de-reference, nested-table, varray, etc.

There is a significant feature of ORSQL, we call it *object navigation*. Utilizing reference type (OID), the attributes $A_1, \dots, A_n$ in one object table T_1 , can be presented

by the attributes of another table T_2 $B_1, \dots, B_n$, if $B_1, \dots, B_n$ reference to the object type of table T_1 .

For example, object table *Country_tab* is created according to object type *Country.t* which has two attributes: *Name* and *Population* (see Subsection 2.2.5). Object table *Province_tab* has attribute *Country* which references to the object type *Country.t*. The following ORSQL statement shows that attributes *name*, *population* of object table *Country_tab* can be represented by attribute *Country* of object table *Province_tab*:

Select p.country.name, p.country.population from province p.

2.2.8 Chapter Summary

In the discussion above, we have concentrated on how the object-relation data model extends the relational model. Although these object-relational database features enable it the capability of handling complex data, the object-relational database cannot exert its advantages if there is not a well-defined conceptual model to model the complex data meaningfully. On the other hand, the complication of the object-relational database design needs a conceptual model that can easily be mapped to the database schema.

Chapter 3

Related Work

In accordance with the objectives of this thesis, the related work will be examined in relation to two aspects: conceptual data models and natural language interface to databases. Section 3.1 introduces two conceptual data models: ER and EER models, which are today's most popular conceptual data models. Section 3.2, introduces several approaches for natural language interfaces to databases.

3.1 ER and EER Conceptual Data Model

A conceptual data model is intended to model real world applications in a natural way. The **Entity-Relationship (ER) model** was originally proposed by Peter Chen in 1976 [10], as a way to unify the network and relational database views. It is a conceptual data model that views the real world as entities and relationships. A basic component of the model is the Entity-Relationship diagram which is used to visually represent data objects. The **enhanced-ER(EER)** model includes all the modeling concepts of the ER model. In addition, it includes the concepts of **subclass** and *super-class* and related concepts of *specialization* and *generalization*.

For the database designer, the utility of the ER or the EER model is advantageous for three main reasons:

- The same constructs used in the ER model can easily be transformed into relational tables.
- It is simple and easy to understand with only a minimum level of training required. For this reason, the model can be used by the database designer to communicate the design to the end user.
- In addition, the model can be used as a design plan by the database developer to implement a data model in a specific database management software.

3.2 Natural Language Interfaces to Databases

A natural language interface to databases is a system that allows the user to access information stored in a database by using some natural language (e.g. English).

Since the early 1960s, much of the research on NLIDB has been motivated by its potential use of communicating with databases. Early natural language systems include those such as LUNAR [47] and SHRDLU [46]. LUNAR is a natural language interface to a database that contained chemical analyses of moon rocks, and resolved pronouns by consulting the predefined entities. SHRDLU allows a user to manipulate a block world by using English instructions. The system possesses a notion of context and could handle pronoun and some definite noun phrase resolution, as well as ellipsis.

In the 70s when linguistics and natural language processing focused for a long time primarily on the syntactic structure of sentences, [48] shows that good representations are the key to good problem solving. The *semantic networks* described in [48] can

convey the natural semantics. The *Conceptual Dependency representation* described in [40] claims that semantics representation is critical, which argues that even if one's goal is to build intelligent machines, it is a good first step to model how humans perform the task.

From the approach point of view, the research conducted on NLIDB can be categorized as grammar-based and non grammar-based [2, 19, 38, 42]. There are three main approaches in the grammar-based systems.

- The statistical approach constructs a probabilistic grammar which requires supplying a syntactic parser tree, as well as a semantic representation for each training sentence, and also requires hand-crafting a set of contextual features on which to condition the parameters of the model. The *statistical approach*, such as mapping airline-information queries into SQL [30] using a N-Gram vector for capturing lexical context [11], and a probabilistic decision-tree method for the same task described in [26].
- The logical approach is another important approach. For example, using relational learning to learn a logic-based semantic parser was described in [49]. The logical approach uses a deterministic parser which may suffer from robustness problems.
- Also, integration of the *statistical* and *logical* approach is also one approach presented in [44], where a system, WOLFIE, acquires a semantic lexicon from a corpus of sentences paired with semantic representations.

Generally speaking, these traditional grammar-based approaches can make the system grow incredibly complex as the rule structure changes dramatically for each language.

Non grammar-based systems, such as pattern matching systems, have been proposed as an alternative to grammar-based systems [21, 11]. These systems often use templates to match certain words and phrases instead of depending on grammar.

Because of the difficulty of developing a large body of machine-interpretable knowledge based on human linguistic behavior, natural language interfaces to databases have not gained the expected rapid and wide commercial acceptance [2, 9]. There are two good reasons [29, 38] to explain why:

1. Lack of portability. Existing system can only cope with questions referring to a particular knowledge domain. So when we transfer an existing NLIDB to different knowledge domains, it is very expensive or sometime impossible to do so.
2. Difficulty of usability. Existing NLIDBs are brittle and users are often frustrated using them. This is because of the coverage mismatch – this means that users cannot often distinguish between the limitations of the system’s conceptual coverage and the system’s linguistic coverage.

Chapter 4

Category Model

The category model adopts the natural view that the real world consists of interrelated objects that can be classified into various categories and their relationships. In this model, categories and category relationships are the fundamental concepts through which information resources are modeled. It is important that the category model is built independently from the data resources, and captures the semantic meanings of the data in a lasting manner. Therefore, the schema of the database is *stable* even when the information resource changes over times. The model incorporates the important semantic meaning about real world entities that is useful for distinguishing the ambiguity of natural languages. The model is based on set theory, relation theory and methodologies of the object-oriented paradigm.

In this chapter, we define the category model and demonstrate its properties. We begin our discussion in Section 4.1 by introducing objects and attributes in the category model. Section 4.2 introduces the concepts of the category. Section 4.3 presents the category model diagram, a tool for system analysis. Section 4.4 shows the category hierarchy. The relationship in the category model is introduced in Section 4.5. Section 4.6 provides an example of data modeling using the category model.

Section 4.7 presents the features of the category model.

4.1 Objects and Attributes

In the category model, objects represent real world physical and conceptual entities. A specific person or university is an example of an object. Objects have attributes through which they are related to each other. An object *attribute* is a named property of the object that describes the value held by the object. There are four kinds of attribute values in the category model:

1. Atomic values such as strings ‘Carleton University’, ‘Canada’, integers 10, 12, real numbers 1.2, 0.1.
2. Tuple values such as *Contact_information*[*Phone* $\Rightarrow$ 613-2312465, *Fax* $\Rightarrow$ 613-2321457, *Email* $\Rightarrow$ info@carleton.ca], *Family*[*Husband* $\Rightarrow$ Bob, *Wife* $\Rightarrow$ Alice], *Teaching*[*Instructor* $\Rightarrow$ Bob, *Course* $\Rightarrow$ {*Course*₁, *Courese*₂, *Course*₃}].
3. Object identifiers (OIDs). An OID is the property of an object which distinguishes it from all others and is used to reference the objects. OIDs and objects have an immutable relationship. In other words, one OID represents one object and one object has one OID. Thus, one object can be represented by its OID.
4. Set values such as {}, {*Ontario*, *Quebec*, *Alberta*},
{*Teaching*[*Instructor* $\Rightarrow$ Bob, *Course* $\Rightarrow$ {*Course*₁, *Courese*₂, *Course*₃}]} }

Representing a specific real world entity by which kind of attribute values depends on the nature of the problem at hand. There may exist several correct solutions. In order to demonstrate how to use these kinds of attribute values to represent the real world objects, we use Figure 4.1 to illustrate the attributes defined on object *Carleton*

Figure 4.1: Category Model Example

University. The value of attribute *Name* is a string. Attribute *Contact_information* is a tuple value which represents contact information, *Telephone*, *Fax*, and *Email*. *Departments* are defined as objects. To present departments within *Carleton University*, we define the attribute *Departments* which is a set value. The value *OID1* represents an OID of object *School of Computer Science*. *OID2* and *OID3* represents the OIDs of the objects *Department of Physic* and *Department of Chemistry* respectively.

In the category model, every object has an attribute *Name*; this is obvious because every entity in real world has a name. Attribute *Name* often can be used to identify an object, but this is not always true. For example, *Eddy* may be either a person's name or the name of a street. Thus, in the category model, we consider *Name* as a fundamental or default attribute of an object, while the OID is considered the key from which we can identify objects.

The attributes of an object in the category model have two features:

- The domain of the value type is extended by the tuple value, set value and OID, rather than a simple or atomic attribute type which cannot be divided.
- An attribute can be a nested relation. The nested relation extends the flat relational by relaxing the first normal form assumption to allow for the modeling of complex objects. An example of a nested relation is attribute *Departments* of object *Carleton University* provided in Figure 4.1.

4.2 Categories

Although any information systems deal with information about real world objects, the concept of category occurs naturally in people's mind, where it is used to distinguish entities. For example, in a library or a grocery store, *categories* help us find the books or food items we are looking for.

The definition of *category* according to Webster's New World Dictionary is

Category is a general class of ideas, terms, or things that mark divisions or coordinations within a conceptual scheme.

The concept of category, as utilized in the category model, is the same as this definition. Category represents a collection of objects which can be distinctly identified. For example, the category *Person* represents a collection of persons. *University* is also a category, which represents a collection of universities. Category in the category model provides a mechanism for sharing across similar objects with common semantics.

There is an *instance-of* relationship between objects and categories. In Figure 4.1, object *Carleton University* is classified into category *University* and *School of Computer Science*, *Department of Physics* and *Department of Chemistry* are objects of category *Department*.

A category has attributes which are the common attributes of its objects. In Figure 4.1, the attributes of category *University* are *Name*, *Contact information* and *Departments*, which correspond to the common attributes of objects in this category.

4.3 Diagram Representation

Based on UML [36], we provide a set of notations to represent the category model in diagram form. This diagram spans the development cycle; the models that are developed during analysis carry forward to the database design and implementation phases. Since UML has the extension mechanisms that can apply to any modeling elements [36], we extend the UML terms and notation according to the contents of the category model as follows:

- The extension from class role to category role. Since we use term *category* to represent a set of objects in the category model, we extend class diagram in UML to category diagram in the category model.
- The extension of attribute types. Since UML is derived from the OO programming language. The data types it represents are different from the category model. We extend attribute types, such as tuple and nested relation, for the usages in the category model.
- The representation of relationships. We use two attributes in two categories to represent the traversal paths of the relationship.

We denote a category by a box with the category name on the top of portion of the box; the remaining portion of the box lists attributes of a category. According to the object attribute values, the category attribute types are *atomic*, *OID*, *tuple*, and *set*. The notation of attribute types is presented as follows:

- If the attribute type is atomic, it is represented as String, Number, Date, etc.
- If the attribute type is tuple, the type is represented by the name of the tuple.
- If the attribute type is OID, the type is represented as: *ref category*.
- If the attribute type is a set, the type is represented as: $\{ \text{attribute type} \}$.

Figure 4.2 illustrates category *Person* and its attributes. Attributes *Name*, *Age*, *Gender* are atomic types; the type of attribute *Contact information* is represented by the tuple name *Contact_information*; The types of attributes *Spouse* and *Children* are OID and set type.

Figure 4.2: Category Person

Figure 4.3: Category Hierarchy Example

4.4 Category Hierarchy

Categories may have a hierarchical structure. That means one category can have sub or super categories when the groups of objects they denote are subsets or supersets of the corresponding groups. A super-category holds common attributes whereas the sub-categories inherit the attributes of their super-category and can add additional attributes specific to the sub-category. For example, besides its specific attribute *salary*, category *Employee* inherits the attributes *Name* and *Gender* from its super-category *Person*. There is an *ISA* relationship between super-category and sub-categories, which organizes categories by their similarities and their differences. Similar to the object-oriented data model, category hierarchy can have multiple levels relationships.

Figure 4.3 shows a category hierarchy in the upper layer and objects classifications from the upper layer to the lower layer. In the category hierarchy, the category *Person*

has two sub-categories *Student* and *Employee*. The category *Employee* has the sub-categories *Professor* and *Staff*.

Now consider classifications that link categories in the upper layer to objects in the lower layer. The category *Professor* has the direct objects *John*, *Eric*, and *Bob* and category *Staff* has the direct object *Adam*. These direct objects are indirect objects of the category *Employee*.

The characteristics of the category hierarchy allow the category model to support *inheritance* and *polymorphism* mechanisms as described in the object-oriented data model [1, 8, 16, 25, 35]. *Polymorphism* is critical, because it allows many categories derived from the same category to be treated as the same categories. For example, we can treat the category *Professor* as the category *Employee*, or as the category *Person* depending on our needs at a given time.

Utilizing object classification and category hierarchy, the information can be modeled more meaningfully.

Figure 4.4 presents a category inheritance example using the category model diagram. A large hollow arrowhead denotes category inheritance. The arrowhead points to the super-category. The super-category may be directly connected to its sub-categories, however it is normally preferred to group sub-categories as a tree. In Figure 4.4, category *Object* is the super category of all other categories. It only has one attribute *Name*, so all other categories inherit this attribute.

4.5 Relationships in the Category Model

Relationships in the category model are defined between categories. Relationships may be one-to-one, one-to-many, or many-to-many, depending on how many objects

Figure 4.4: Inheritance in the Category Model

of each category participate in the relationship. For example, *marriage* is a one-to-one relationship between two objects of the category *Person*. A person can have a one-to-many *parent_of* relationship with multiple children. Teachers and courses typically participate in many-to-many relationships.

In the category model, categories are interconnected by their relationships. Unlike ER model [10], in the category model, relationships are represented as attributes of the categories. In particular, if two categories, C_1 and C_2 have a relationship, R_1 , then relationship R_1 has two *traversal paths* – the path from C_1 to C_2 and the path from C_2 to C_1 . Attribute A_{C_1} in category C_1 and attribute A_{C_2} in category C_2 represent this two traversal paths.

To present category relationships in the category model diagram, we use a line with arrowhead to present one of the traversal paths of a relationship. The line starts at the corresponding attribute, while the arrowhead of the line points to the

corresponding category.

To represent one-to-one, one-to-many, or many-to-many relationships in the category model diagram, we define *Multiplicity* as the number of objects of one category that may relate to a single object of another associated category. Figure 4.5 summarizes multiplicity combinations. The star above the arrowhead means *many (zero or more than one)* multiplicity, for example a country may have many *provinces*; a hollow arrowhead denotes *zero or one* multiplicity, for example one paper may or may not be published at an international conference; the arrowhead without any symbols above means *exactly one*, for example, a province belongs to one country. Symbols above the arrow head indicate other specific multiplicities, such as $1..*$ or $3..5$. We may also list several ranges such as $2..4$, $8..21$, $32..40$ multiplicity. We may specify that the collection of objects for *many* multiplicity is ordered.

Figure 4.5: Notation for Multiplicity in the Category Model

Figure 4.6 illustrates categories *Country*, *Province*, *City*, and *University* and their relationships: one country may have several provinces and one capital city; one province belongs to one country and may have several cities; a city belongs to one province and may have several universities; one university is in one city.

Figure 4.6: Relationships in the Category Model

4.5.1 Dependency Relationships

In the category model, the existence of some categories depends on other categories. For example, category *Department* are related to *University* by means of an *dependency relationship*: each university can have many departments and conversely, the existence of a department depends on the existence of a university. We call category *University* a parent category and *Department* a dependent category.

It is important to note that the dependency relationship we present herein is not equal to the *Aggregation* or *Part-off* relationship [14, 20, 39, 41] in the object data model. Some aggregation relationships are the dependency relationship and some are not. For example, hard disks are part of a computer, but if a computer is disassembled, hard disks still exist as objects. Departments, on the contrary, cannot exist independent of the university of which they are part.

There is a consistence constraint between the parent category and the dependent category. For example, deleting a university implies the deletion of its departments.

The dependency relationship is the key to semantic integrity checking. In other words, the dependency relationship allows to track and solve inconsistencies in the category model.

We use a broken line with arrowhead to represent a dependency relationship. A line starts at a dependent category and the arrowhead points to a parent category. Figure 4.7 shows the dependency relationship between category *University* and *Academic Unit*.

Figure 4.7: Existence Dependency Relationship

The dependency relationship can be transferred by the category inheritance. That means if a super-category is a dependent category, its sub-categories also have this dependency relationship with the parent category. In Figure 4.7, category *Department*, *Faculty*, and *School* have the dependency relationships with category *University*.

4.6 Example

There are four steps in modeling an application using the category model:

1. Identify the categories of interest.

2. Identify the relationships among the categories.
3. Identify the semantic information in the relationships, such as whether a certain relationship is one-to-one or one-to-many, then decide which kind of attribute types should be used to represent them.
4. Identify the attributes by various value types.

Utilizing the category model, we are able to model some real world applications with complex relationships. To demonstrate this issue, we illustrate an example of modeling university and its academic units. From our investigation, it is apparent that the organizational structure of universities varies a lot. We illustrate the varieties as follows:

- Case 1– A *University* includes several *faculties* such as *Faculty of Science*, *Faculty of Arts and Social Sciences*, *Faculty of Engineering*, etc. A faculty, for example *Faculty of Science*, includes several departments such as *Department of Computer Science*, *Department of Physics*, and *Department of Chemistry* and so on.
- Case 2– In some universities, there is an academic unit – *school* situated between the *faculty* and *department* levels – that is, departments are included in schools and schools are included in faculties.
- Case 3 – Some universities treat schools and department at the same level, or as the same unit.
- Case 4 –Some universities don't have the *faculty* academic unit, only departments.

```

classDiagram
    class Person {
        gender : String
        Spouse
        Contact information:
        Contact_information_t
        ..*
    }
    class Province {
        country
        Cities
    }
    class City {
        Province
        Universities
        Website: URITY PE
        ..*
    }
    class Academic_unit {
        SubUnit
        University
        Professors
        ..*
    }
    class University {
        Academic Unit
        President
        City
    }
    class Staff {
        AcademicUnit
        Title :String,
        ..*
    }
    class Professor {
        Academic Unit
        Research: String
        ..*
    }
    class Faculty {
    }
    class School {
    }
    class Department {
    }

    Person <|-- Staff
    Person <|-- Professor
    Academic_unit <|-- Faculty
    Academic_unit <|-- School
    Academic_unit <|-- Department

    Person --> Person : self
    Province --> City : Ref
    City --> Province : Ref
    City --> City : Ref
    City --> Academic_unit : Ref
    Academic_unit --> City : Ref
    Academic_unit --> Academic_unit : Ref
    Academic_unit --> University : Ref
    University --> Academic_unit : Ref
    University --> University : Ref
    Staff --> Academic_unit : Ref
    Professor --> Academic_unit : Ref
    Professor --> Professor : Ref
    Professor --> Academic_unit : Ref
    Faculty --> Academic_unit : Ref
    School --> Academic_unit : Ref
    Department --> Academic_unit : Ref
    University --> University : Ref
    
```

Figure 4.8: Data Modeling Example

Figure 4.8 illustrates our category data modeling approach. First, categories *University* and *Academic unit* are defined and have the relationship: one university may have several academic units and one academic unit belongs to a university. *Subunit*, an attribute of category *Academic unit*, references to its own category, *Academic unit*. This creates a *circular reference* (see Chapter 2). This circular reference is important because it allows the subunits of one academic unit are also in category *Academic unit*. Then, because categories *School*, *Department*, and *Faculty* inherit from *Academic unit* category, each of these three categories also has attribute *subunit* which

references to category *Academic unit*. Utilizing polymorphism, categories *School*, *Department*, and *Faculty* can be treated as category *Academic unit*. Thus, each of these three categories can be referenced by the attribute *subunits* of other categories. In other words, the subunits of *Faculty* can be *Department* or *School*; the subunit of *School* can be *department*. So, the problems of all the possible organizational cases are indeed resolved by this modeling.

4.7 Database Design

This section discusses how to map category model to the object-relational database, along with the examples.

4.7.1 Object-relational Database Design using the Category Model

Mapping from the category model to the object-relational database schema is straightforward:

1. The mapping from the category attribute types to the corresponding object-relational database types is shown in Table 4.1.
2. We use one object type in the object-relational database to represent one category in the category model.
3. The inheritance of the category is represented by the inheritance of the object type in the object-relational database.
4. To represent the relationship R_1 of categories C_1 and C_2 , a pair of attributes A_{T_1} and A_{T_2} are defined in object types T_1 and T_2 respectively. T_1 and T_2

The Attribute Types in the Category Model	The Types in the Object-relational Database
Atomic value Such as Real, Integer, String	Primary types Such as Real, Integer, Varchar
Tuple	Object type
Object Identifier (OID)	Reference
Set	Varray
	Nested table

Table 4.1: Types Mapping

represent C_1 and C_2 . A_{T_1} references to objects type T_2 , while A_{T_2} references to object type T_1 . The attribute with a nested table or varray of reference type represents a one-to-many relationship, while the attribute with a reference type represents a one-to-one relationship.

5. The names of object types and attributes are the same as the names of the categories and attributes in the category model except for the suffix *_t*. “*_t*” is the suffix of the object type.

Because an object type is a template, defining it does not result in storage allocation. Determining whether to store a particular object type to an object table depends on whether it simplifies the query. We use one object table to store one object type that has no inheritance relationships with other object types. For example, an object table *University_tab* is created according to object type *University_t* (see Figure 4.8). For object types with inheritance relationships, we use object tables to store these object types, which represent the *top-level* ontologies [18], describing very general concepts such as *person*. This approach is feasible because utilizing polymorphism, tuples with subtypes can be stored in the object table with the super-type.

For example, we use object types *Academic_unit_t*, *Department_t*, *School_t* and *Faculty_t* to represent the categories *academic unit*, *Department*, *School* and *faculty* (see Figure 4.8). Object table *Academic_unit_tab* is created according to object type *Academic_unit_t*. Since object type *Academic_unit_t* is the super-type, tuples with type *Department_t*, *School_t* and *Faculty_t* can be stored in table *Academic_unit_tab*. Table 4.2 illustrates this approach which is easy for the query concerning on types *School_t*, *Department_t*, or *Faculty_t*.

Academic_unit_tab
SCHOOL_T('School of Business', , 'School in University of Alberta')
DEPARTMENT_T('Department of Agriculture, , 'Department in Faculty of Agriculture, University of Alberta')
DEPARTMENT_T('Department of Geography' , , 'Department in Faculty of Arts, Memorial University')
FACULTY_T('Faculty of Medicine', , 'Faculty in University of Calgary')

Table 4.2: Tuples with different object types stored in one object table

4.7.2 Mapping Examples

According to the sample schema shown in Figure 4.8 at Section 4.6, we map category *Person* to object type *Person_t*; attribute *Gender* in category *Person* is mapped to attribute *Gender* in object type *Person_t* and category attribute type *String* is mapped to attribute type *Varchar*. Since attribute *Contact Information* in category *Person* is a tuple value, we map it to object type *Contact_info_t*. The following schema shows the corresponding ORSQL code for the creation of object type *Person_t* and its attributes.

```
create or replace type Contact_info_t
    (Phone varchar2(40),
```

```

Fax varchar2(40),
Email varchar2(40));

```

```

create or replace type Person_t
(Gender varchar2(6),
Spouse ref Person_t,
Contact_Information Contact_Info_t,
.....) not final;

```

In Figure 4.8, category *Professor* and *Staff* inherit from category *Person*. To map this inheritance relationship, object types *Professor_t* and *Staff_t* are defined to inherit from object type *Person_t*. The following ORSQL code demonstrates the creation of object types with inheritance relationships. The reserved word *under* means one object type inherits from its super-type. Attributes *Gender*, *Spouse* and *Contact_Information* in object type *Professor_t* are inherited from object type *Person_t* and at the same time, object type *professor_t* can have its own attributes, for example the varray type attribute *Degree*.

```

create or replace type Degree_v as varray(5) of varchar2(100);
create or replace type Professor_t under Person_t
Research varchar2(2000),
Degrees Degree_v,
.....);

```

In Figure 4.8, there is a relationship that an academic unit may have several professors and a professor may work in an academic unit. To represent this relationship, we first define an object type *Professor_nt* which is a nested table of references referencing to object type *Professor_t*. Then, we define object type *Academic_unit_t*; its attribute

Professors is defined as object type *Professor_nt* to represent that an academic unit may have several professors. Similarly, to represent that one professor may work in an academic unit, we define attribute *Academic_unit* in object type *Professor_t* to reference to object type *Academic_unit_t*. The following ORSQL code shows the definition of object type *Professor_nt* and *Academic_unit_t*.

```
create or replace type Professor_nt as table of ref professor_t;
create or replace type Academic_unit_t
    Professors Professor_nt,
    .....);
```

From the above discussions, we can see that there is a very close relationship between the category model and the database schema. In our database design, each category has a corresponding object type. The tables in object-relational databases serve to store these object types. The object type inheritance and polymorphism mechanisms make this flexible storage feasible.

4.8 Features of the Category Model

One feature of the category model, most importantly, is its remaining semantic meaning where human understandable content is structured in such a way as to make it machine processable [3, 12, 13, 43]. In other words, the category model provides a common framework for expressing the information, so information can be exchanged between people and applications without loss of meaning. The points supporting this statement are shown as follows:

1. In the category model, the definition of the category is the same as in real world. Thus, the category in the category model retains its semantic meanings.

2. In the category model, relationships among categories are defined according to the relationships among the real world entities.
3. The mapping from category model to database schema does not change the data structure, i.e. the machine are able to interpret the information provided by the category model unambiguously..
4. Without the restrictions of normalization, the attributes and relationships of the category can be organized in the same manner as in real world.
5. Complex value types and the category hierarchy in the category model provide powerful methods to simulate real world entities.

The category model has the wide domain coverage. Every entity in real world is classified into categories by people, so, every entity can be modeled into the category model.

The features of the category model can help the natural language processing distinguish the ambiguity of the natural language.

Chapter 5

Natural Language Patterns

Natural language interfaces to databases allow users to use natural language to express database queries even when users know little about the database structure. However, the natural language expression is so flexible that even a simple query may have various expressions. This creates a challenge for the implementation of NLIDB. For users, the main goal is to query databases in a semantic way. It is not necessary for users to use complex sentence structure as long as they can reach their query goal. Therefore, the user's query needs can be satisfied by providing some natural language patterns which keep the semantic meaning and have the wide domain coverage, while at the same time making it easy for users to construct natural language queries.

In this chapter, we discuss the natural language patterns. Section 5.1 introduces the fundamental concepts through which the patterns are built up. Section 5.2 shows the natural language patterns along with examples. Section 5.3 analyses advantages and limitations of these patterns. Finally, we summarize this chapter in Section 5.4.

5.1 Concepts

The concepts of *category* and *category relationship* are fundamental in the category model. Utilizing these concepts, we have built several natural language patterns. Since the category model considers real world applications as a set of categories, the function of these patterns is to query objects in the category. The natural language patterns can be abstracted as:

Selected attribute + Topic category + Conditions.

The topic category refers to the theme of the query. For example, the query *Find the name, websites of professors in Carleton University whose research includes database* has topic category *professor*. We call the specific attributes of a topic category *selected attributes*. For example, *name* and *website* are selected attributes. When selected attributes are omitted, the query implicitly includes all attributes of the topic category. *Conditions* refer to the elements in the query modifying the topic category. For example, *Carleton University* and *research includes database* are *conditions* of topic category *professor*. We call *University* a *condition category*, since it is a category modifying the topic category. Condition categories are connected by prepositions such as *in, at*, etc. We call this preposition phrase a *category phrase*. The implicit relations among these *conditions* are *and*. According to our patterns, the *conditions* for the topic category are found in three places:

- (1) The values of the condition categories. For example, in the query above, *Carleton University* is the value of condition category *University*.
- (2) The values of attributes in the topic category. For example, *database* is the value of attribute *research*.
- (3) The value of the topic category. Consider the query *Find city Ottawa*: the value

Ottawa is the condition of the topic category *City*.

5.2 Patterns

5.2.1 Pattern 1

This pattern is used for querying objects in the topic category and the conditions may come from different categories. We formulate Pattern 1 as follows:

[Select attributes] + Topic category + [Value] + [Category phrase] + [Whose clause][With phrase].

The components in [] are optional. Two elements concatenated by | means that one is an alternative of another.

The category phrase is formulated as follows:

Category phrase: $P + \text{Category} \oplus \text{Value} + [P + \text{Category} \oplus \text{Value}]$.

P represents a preposition. Two components concatenated by $\oplus$ means that at least one of them must appear. So, category phrase allows for the *value without category*, although it is very easy to identify the meanings of values through the category. This extension expands the pattern's ability to express queries naturally (e.g. *universities in Ottawa*). On the other hand, such an extension raises a challenge for linguistic analysis: How can the category of words be identified in their context? We introduce our approach in the next Chapter.

The whose clause contains the attributes of the topic category and their values. It is formulated as follows:

Whose clause: *Whose* + Attribute + *OP* + Value + [and|or + Attribute + *OP* + Value].

Comparison operator *OP* is one of *is*, *is not*, *are*, *are not*, *include(s)*, *exclude(s)*, *is(are) more than*, *is(are)less than*, *is (not) over*, *are (not) over*, *is(are) not more than*, *is(are) not less than*.

The whose clause has an alternative expression – *With phrase*:

With phrase: *With* + Attribute + [*OP1*] + Value.

The *OP1* is one of these: *over*, *below*, *under*, *more than*, *less than*, *no more than*, *no less than*, *greater than*.

Now, we illustrate some examples constructed according to Pattern 1. We start with the query: find *country Canada*, a very simple but very common query. Excepting the word “*find*”, there are two words in this query. One is considered as the topic category *country*, and the other is the value *Canada*. This kind of query – *Topic category + Value* is widely used, for example *Professor Smith Morrison*, *Lawyer John R. White*, *Carleton University*, *Government Canada*, *Ottawa city*, *Course 95.590h*, and so on. This is because in real life, we can not always identify one object only by its name. For example, if one asks for “buffalo”, a variety of answers may be given, such as animal buffalo, buffalo city, buffalo school, person by this name, etc. Therefore, *Category + Value* can be used to identify the object’s meaning, despite the ambiguity of natural language.

Pattern 1 allows the addition of selected attributes to the topic category. For example, in the query: find *population of country Canada*, *Population* is the selected attribute from the topic category *country*. This kind of query is also very common, for example *tuition fee of Carleton University*, *Contact information of professor Smith*

Morrison, Homepage of city Ottawa, etc.

To query the selected attributes, we also can add some interrogatives into queries to get some variants of Pattern 1. For example, queries: *what is the location of Carleton university?* *Where is Carleton University?* imply querying the selected attribute *Address* of topic category *University*. They are equal to the query: *find the address of Carleton University*. These kinds of variants make the structures of queries flexible and easy for users to understand.

According to Pattern 1, we can use condition categories and its values to modify the topic category. In the query: *find universities in province Ontario*, the condition category *province* and its value *Ontario* limits the location of *universities*. There are also many queries that can be expressed in this way: *faculties in Carleton University*, *professors in city Ottawa*, *professors in the school of computer science at Carleton University*, *cities in Country Canada* or *Cities in Canada* and etc.

The *whose* clause is used to construct more complex queries. It focuses on the attributes of the topic category. Logical operators such as *and*, *or*, and *not* allow users to assemble various Boolean conditions; the comparison operators, such as *more than*, *less than*, and *over* enable the query to have calculation and comparison functions. Here are some queries with the *whose* clause: *find countries whose population is more than 1 million and GDP is not less than 2000000*; *find universities whose programs include MBA*; *find universities whose programs include MBA and name is not Carleton University*; *find universities whose graduate tuition is less than 2120*.

We can also use the *with* phrase to replace the *whose* clause, making the query shorter. Here are some examples: *find countries with population over 30 million*; *find universities with graduate tuition under 2200*; *find professors in Canada with name John White*.

Combining the above features, we can flexibly construct the query according to Pattern 1, for example find *the contact information of professors in Ontario whose research includes database, find universities in Canada whose programs include MBA and graduate tuition is less than 2120, etc.*

We also can use *Are there?* to construct another variant of Pattern 1. This pattern focuses on the topic category, and excludes the selected attributes. Here are some queries: *Are there any professors in Carleton University whose research includes Database? Are there any universities in Ontario?*

5.2.2 Aggregation

The aggregation is very important, as it provides powerful calculation and comparison methodologies for user queries. In order to represent the aggregation functions in patterns, We introduce five *aggregation keywords*:

1. *Maximal, Maximum* – modifies an attribute of the topic category and calculates its maximal value.
2. *Minimal, Minimum* – modifies an attribute of the topic category and calculate its minimal value.
3. *Average* – calculates the average for a attribute of topic category.
4. *Number of* – counts the number of objects in the query.
5. *Total (of)* – calculates the sum value of the selected attributes.

Aggregation keywords 1-3 are used in the whose clause or with phrase as the special values. Integrating them, we build Pattern 2 which can calculate the maximal, minimal, or average value of the attributes in the topic category:

Pattern 2: Topic category + [Category phrase] + Whose clause |With phrase

Following are some queries according to this pattern:

- Find universities in Canada whose tuition fee is minimal.
- Find universities in Ontario whose student enrolment is maximal.
- Find countries with minimal GDP.
- Find countries whose GDP is over average.
- Find countries with maximal population.
- Find countries whose GDP is minimal or population is minimal.

The aggregation keyword *number of* is treated as an aggregation function which counts the number of objects in the topic category. We use this aggregation keyword or its alternative representation *How many* or *How much* to build Pattern 3:

Pattern 3: *Number of*|*How many*|*How much* + Topic category + Category phrase $\oplus$ + Whose clause |With phrase

Following are some queries according to this pattern:

- Find the number of universities in Canada.
- Find the number of countries whose GDP is over average.
- Find the number of countries with population more than 1 billion.
- How many professors in the Physics department at Carleton University?
- How many universities in Ontario?

- How many professors in Ottawa whose name is John White?
- How many cities in Ontario with population over 0.3 million?
- How many departments in Carleton university?

We integrate aggregation keyword *total (of)* to build Pattern 5 which can calculate the sum of the selected attributes:

Pattern 4: Total (of)+ Selected attribute+ Topic category + [Category phrase] +[Whose clause | With phrase]

Some queries according to this pattern are shown as follows:

- Find the total students of Carleton University.
- Find the total graduate students of Universities in Ontario.
- Find the total undergraduate tuition fee of Carleton University.

Pattern 1-4 are all constructed by the topic category, selected attributes, condition categories, whose clause, or with phrase. The aggregation keywords 1-3 can be considered as the special values of attributes in the topic category; the aggregation keywords 4-5 can be considered as the special selected attributes. From this point of view, Pattern 2-4 can be translated into Pattern 1. So, by handling the parsing action of Pattern 1, we are able to handle other patterns' parsing actions as well.

5.3 Advantages and Limitations

From the discussion above, we can summarize the advantages of the patterns as the follows:

- The knowledge domain coverage of patterns is wide. Since the category model focuses on simulating entities and relationships in real world, it has a wide domain coverage. The patterns are built based on this model. As such, these patterns can be applied in various knowledge domains as long as these domains can be described by the category model.
- The patterns have the capability of constructing complex queries.
- The patterns allow comparison, calculation and inference in the query.
- Users can select multiple patterns to express a query.

On the other hand, these patterns also have the negative aspect. The main limitation is that although these patterns provide very powerful ways for users to construct their queries, they cannot cover all possible queries structures due to the flexibility and diversity of natural language expression. Further studies on expanding the patterns' structure to meet more complex queries should be pursued.

5.4 Chapter Summary

This chapter presents the natural language patterns that are the foundation of our quasi-natural language interface. Several fundamental concepts and terms are introduced in this chapter. We also discussed the capabilities of these patterns to construct queries by illustrating how they are used.

Chapter 6

Implementation of the Prototype System

The prototype system provides a quasi-natural language interface for users to query the data gathered mainly from the web and structuralized by the category model. In this chapter, we discuss the design and implementation of this system.

This chapter is organized as follows. Section 6.1 introduces the abstract models in the natural language interface to databases. Section 6.2 introduces the system architecture. Section 6.3 presents our approach for identifying the meaning of a keyword. Section 6.4.2 presents the alias matching approach. Section 6.5 introduces the parsing processes. Section 6.6 presents examples to demonstrate the parsing processes. Section 6.7 introduces the user interface of this system.

6.1 Abstract Models

Generally, there are three levels of models involved in a NLIDB, which are users' linguistic model, a conceptual data model and a actual data model [29] such as the

relational data mode and object-relational data model. The task of NLIDB is to map the users' linguistic model to the actual data model. However, the distance between them is very far, because the actual data model does not contain any semantic information and linguistic models are flexible and varied. An intermediate representation – the conceptual model is therefore introduced into most systems. The linguistic model is first mapped to the unambiguous conceptual model and then makes the actual data model definite from the conceptual model. The relationships between these models are illustrated in Figure 6.1.

Figure 6.1: Abstract models in NLIDB

Although ER models are also available to express the conceptual model, they have been developed for the relational database design and are not capable of handling complete semantics. As discussed in Chapter 4, the category model incorporates the important semantic meanings about real world entities and structures the contents in a object-relational data model processable way. In other words, the category model, as a conceptual model, not only has semantics to cover parts of linguistic models, but also is easily mapped to the object-relational data model (see Figure 6.1). Thus, the category model is more powerful than the ER model in terms of semantic expression.

6.2 System Architecture

Figure 6.2: System Design

Figure 6.2 shows our system design which is the four tiers structure. The client tier consists of two parts: dynamic web pages containing various types of markup language (HTML, XML, and so on), which are generated by JSP and Severlets running in the web tier, and a web browser, which renders the pages received from the web server. The web tier contains JSP and Severlets programs which dynamically process requests and construct responses, while at the same time using web server (Apache) to communicate with the client tier and the application tier. The application tier contains the application server developed using Java and Javabean, which solves the needs of the parsing. The application tier exchanges data with the web tier and database tier. The database server (Oracle 9i) stores data and contains some database level applications such as triggers and stored procedures that are developed using PL/SQL and SQL.

Figure 6.3: System Architecture

Figure 6.3 shows the architecture of our system. Users' queries are firstly accepted by the *Keyword Processor* which determines one word's meaning in the context. It includes two sub-procedures: *Keyword Search* identifies the word possible categories and *Category Matching* determines the ideal category. After the conversion of the queries according to patterns (see Chapter 5), natural language processing begins. *Natural Language Processor* includes: *Alias Matching Manager* handling multiple words with the same meaning, and *Natural Language Parser* translates natural language queries into ORSQL. DBMS accepts the ORSQL and retrieve data from the database. In the following Sections, we introduce the above implementation issues in the detail.

Figure 6.3 also shows that before the high structured information is stored in the database, *Extractor* extracts potentially useful information from the web pages

according to the category model. *Extractor* is one of critical parts in our project. Although this thesis does not deal with the implementation of *Extector*, A brief overview of *Extractor* is given because it is related to processors. *Extractor* aims to extract relevant information from unstructured or semi-structured data while the processors mentioned early aim to select relevant information. *Extractor* has three subtasks:

1. Resource locating: According to the intended contents, retrieve data in HTML.
2. Information conversion: according to the specific schema in the category model, select the intended information and convert it to XML through XSLT.
3. Database Population: translating XML formation into ORSQL to insert data into database.

6.3 Identifying the meanings of Keywords

If we can get rid of category *City* in the query: find *universities in City Ottawa*, this query becomes more natural to users. This idea raises a question: How to determine a word's meaning? Due to the ambiguity of natural language [5, 11], determining the meaning of a word is really a challenge in the linguistic analysis area, because one word may have multiple meanings in different contexts. For example, word *Ottawa* in query *Find universities in Ottawa* is obviously a city name, but word *Ottawa* itself may also be the name of a person or the name of a product. According to the category model, we determine the categories the word belongs to, thus the meaning of the word can be determined. Before discussing the algorithm for identifying words meanings, we need introduce our *keyword search* approach.

6.3.1 Keyword Search

We develop this keyword search sub-system that allows users not only find the matching results but also the category information to help users refine their search. Figure 6.4 shows the search results for keyword “Carleton”. Every object in the category model has an attribute *Name* (see Chapter 4). Our keyword search approach is based on the names of objects.

Figure 6.4: Screen Shot of the Keyword Search

In the category model, different objects are categorized to different categories even if they have the same name. To implement our keyword search, we introduce an object table *objectDictionary* which has three attributes: attribute *keywordName* whose values are names of objects, attribute *Category* representing the category of objects, and attribute *OID* representing the OID of objects. In order to enable

attribute *OID* to reference to all object type, attribute *OID* is assigned to reference to the object type *Object_t* which corresponds to all categories super-category *Object* (see Figure 4.4 in Chapter 4).

So, searching table *objectDictionary*, we can get not only the matching contents but also, most importantly, the corresponding categories and the OIDs through which we can get the corresponding object in corresponding tables. For example, when we search keyword *Ottawa* in table *objectDictionary*, three tuples will be returned along with three categories: *Professor*, *Product*, and *City* and three OIDs referencing to tables: *Professor*, *Product*, and *City*. (See Figure 6.5). Therefore, keyword *Ottawa*'s possible categories are found.

Figure 6.5: Example of different objects with the same name

Oracle 9i supports *index-organized table* which has a storage organization that is a variant of a primary B-tree. Besides storing the primary key column values of an index-organized table row, each index entry in the B-tree stores the nonkey column values as well. In order to improve the speed of search, we create the table *objectDictionary* as an index-organized table. We use several triggers to maintain the consistence between *objectDictionary* and the other tables. When the data operations, such as insertion, deletion, and modification, are applied to any other tables, the

corresponding triggers will be active to maintain the consistence automatically.

6.3.2 Identify the Categories of Keywords

If we can get the *right* category of a keyword, we know the meanings of a keyword. From the above discussion, keyword search approach can get a set of categories. So, the next step is to determine which categories are the best for the keyword.

In real life, people can determine the meanings of a word according to the context of a sentence. Similarly, since a topic category is the theme of a query, the context of this query can be identified, when the topic category is identified. In the category model, categories are interconnected by their relationships (see Figure 4.8). Therefore, the *right* categories are the categories which have the closest relationship with the topic category. To measure the closeness degree between the topic category and candidate categories, we define the priority value P as follows:

- P is the length of the shortest path from topic category to the candidate categories.
- $P = 0$, if the topic category has attributes which directly reference to the candidate categories.
- $P = n$ ($\infty > n > 0$), if there is n categories between the shortest path from topic category to the candidate categories.
- $P = \infty$, if there is no path from topic category to the candidate categories.

Normally, we choose categories with the smallest priority value P as the intent categories. The following is the algorithm for the category matching:

1. Get the topic category.

2. Get the candidate categories by keyword search system.
3. If the numbers of categories are more than one, calculate the priority values and get the categories with the minimum priority value.
4. If there are more than one candidate categories with the same priority value, let users select it.

Now, we illustrate the processes of identifying the category of keyword *Ottawa* for the sample query *Find universities in Ottawa*. First, we get the topic category of this query *University*; then use keyword search to get the candidate categories *City*, *Professor*, and *Product*; calculate the priority values, P_{City} , $P_{Professor}$, and $P_{Product}$: $P_{City} = 0$, since the attribute *City* of topic category *University* references to category *City* (see Figure 4.8); $P_{Professor} = 1$, because there is a category *Academic Unit* between category *University* and *Professor*; $P_{Product} = \infty$, because of no path from *University* to *Product*. Thus, *City* is the category for keyword *Ottawa*.

In this way, we can find a proper category for a keyword. There is also an assumption: the category model should simulate the entities and their relationships in real world as closely as possible. If the category matching is wrong or there are several candidate categories having the same priority value, we also develop a user interface for users to pick or input another category.

In fact, when the category of a keyword is identified, the query can be transformed to *Category + Value* formation. For example, query *Find Universities in Ottawa* is equal to *Find Universities in City Ottawa*.

6.4 Alias Matching Manager

Several different words with the same meaning are called *synonyms* or alias. For example, both *paper* and *publications* refer to the same category. In order to generate ORSQL to query database, we must identify the synonyms of categories and category attributes from queries. *Alias Matching Manager* deals with this problem. The *alias match* approach can be described from two levels:

1. Parsing action level – maps the synonyms into the corresponding table or attribute.
2. Database level – provides synonyms for a table.

Because the regular expression has a simple syntax and can make the text processing more efficient and easier [15], we use regular expressions to map synonyms in parsing action, which is the primary approach for the alias matching. The database level approach is secondary because it only provides synonyms for tables.

6.4.1 Alias Matching Using Regular Expressions

Alias matching is a many-to-one mapping. We use regular expressions to build a list of alias for these tables or attributes that may have synonyms. For example, users may use *prof.*, *prof*, and *professors* to represent table *professor*. The alias matching program uses regular expression “**professor|professors |prof \.|prof**” to identify these words and represents these words by word *professor*. “|” means either one or several options are permitted in a match. The backslash “\” is used to cancel the functions of the reversed characters in the regular expression. “\.” represents “.”. Using regular expressions, we also can easily handle word inflections. For example, the

regular expression “**facult(y|ies)**” can identify word *faculty* and its plurals *faculties*. In regular expressions, elements in () are treated as a unit.

6.4.2 Alias Matching in Database Level

We use Oracle 9i functionality –*Table Synonym* to build synonyms for tables. For example, synonyms *Person*, *People* represent table *Person_tab*. This approach is very efficient, because a synonym is simply an alias and requires no storage. Another advantage is that synonyms hide the identity of the underlying object. If the underlying object must be renamed or moved, then only the synonym needs to be redefined. Applications based on the synonym continue to function without modification. Synonyms can also simplify ORSQL statements, as the ORSQL statement needn’t include the schema that contains these tables. In our system, we create several public synonyms for tables.

This approach also has some limitations:

1. Due to the syntax restriction, synonym can not express some words such as “*prof.*”.
2. It is more difficult to handle word inflections such plurals than the regular expression.

6.5 Parsing

The aim of parsing is to translate a query with *category +value* formation into ORSQL. Since the patterns 2-4 can be translated into Pattern 1 (see Chapter 5). We discuss the parsing of Pattern 1 to illustrate the parsing processes. There are two steps in a parsing action:

1. According to the natural language patterns, extract three elements from the natural language query, which are fundamental to construct ORSQL: (1) topic category, (2) selected attributes, and (3) conditions. The conditions may come from different aspects (see Chapter 5).
2. Translate these elements into final ORSQL utilizing the object-relational database features.

Let's discuss these two steps in sequence.

6.5.1 Getting Three Elements

In order to get these elements, we use the following reserved words in Pattern 1 to determine the sentence structure: *whose, of, at, in, and, or, is, is not, are, are not, include(s), exclude(s), is/are(not) more than, is/are(not), over, is/are(not)under/below, is/are (not)less than*. These words partition the sentence into several parts called *semantic components*. We treat a natural language query as a tree called *Query Tree* in which the reserved words are the parent nodes of each semantic component.

Example 6.5.1 *Figure 6.6 shows how the following query in Example 6.5.1 is represented with the Query Tree:*

Find name, email, website of professors in Carleton University whose research includes database ,

The steps of extracting topic category, selected attributes, and conditions are as follows:

1. Check whether the query matches the requirements of Pattern 1. If so, use the reserved words to partition the query into semantic components. If not, show what cannot be parsed.

Figure 6.6: Query tree for example 6.5.1

2. Parse each semantic component.

- (a) Get the topic category and the value. Several regular expressions (written in Java) are developed according to categories in the category model. To identify topic category, these regular expressions are used to scan the query one by one. According to the patterns, the value of the topic category is in front of the topic category. After topic category is identified, the value of the topic category can be identified.
- (b) Get the selected attributes. After the topic category is identified, then we can locate the selected attributes as they are either all attributes if the selected attributes are null or just the corresponding attributes.
- (c) Get the condition categories. The parser uses the same regular expressions which are used in (a) to determine if condition categories are included. If so, the parser gets these condition categories as they are behind the condition categories.
- (d) Parse the whose clause. In the *Query Tree*, the reserved word *whose* is the root and *and*, *or* are children nodes if they are included. *is*, *is not*, *are*, *arenot*, *include(s)*, *exclude(s)*, *is/are(not) more than*, *is/are(not) over*, *is/are(not)less than*, *is/are(not)under/below* are the grandson nodes. The

attributes of topic category and corresponding values are put at the leaves of this tree (see Figure 6.6). When the parser goes all over the *Query Tree*, it can get the attributes, relation operators, and values. The reserved words *and*, *or* are translated as *logical operators* between two conditions.

3. Determine the relation operators. The relation operator between categories and their value is “=”; relation operators between attributes and values in the whose clause are more complex. They depend on which reserved words between them. Table 6.1 shows reserved words and corresponding relation operators.
4. Combine the conditions from the topic category, condition categories and whose clause with logical operator *and*.

Reserved words in the whose clause	The relation operator
is, are, include(s)	=
is not, are not, exclude	≠
is /are more than	>
is/are not more than	≤
is/are less than	<
is/are not less than	≥
over	>
under, below	<

Table 6.1: Reserved words and corresponding relation operators

We illustrate several examples to demonstrate the parsing procedure shown above. We start with two queries with the same structure: Find *School of Computer Science* and find *name of professors*.

1. Get the topic category. From the regular expressions that are developed to identify the categories, there are two regular expressions corresponding to these

queries: “(?:departments?|schools?) \s+of\s+(\S.*?)” and “(?:professor|professors?|prof\.|prof)(?:(:\s+)(\S.*))?”), which are used to identify topic categories: *department* and *professor*. “\s” represents a space character and “\S” represents a non-space character. Therefore, *Computer Science* is the value of topic category *department* according to Pattern 1.

2. Get the selected attributes. After topic category *professor* is identified, according to Pattern 1, the select attribute is *name*.

Here is an example to demonstrate the processing of extracting three elements from Example 6.5.1

1. Get the semantic components partitioned by reserved words *of*, *at*, *whose*, *includes*. The semantic components before *whose* are: *name*, *email*, *website*, *professors*, *School of Computer Science*, *Carleton University* and the components after *whose* are: *research*, *database*.
2. Using regular expression “(?:professor|professors?|prof\.|prof)(?:(:\s+)(\S.*))?”), identify the topic category *professor*. The value is *null*; the selected attributes are *name*, *email*, and *website*. Using regular expression “(?:university(?:ies)) \s+of\s+(\S.*?)”, we can get condition category *University* and value *Carleton University*.
3. Get *conditions* from the *whose* clause: attribute *research* with value *database*.
4. Combine conditions. Since the value of topic category is *null*, conditions come from the category phrase and *whose* clause: *University*=“*Carleton University*” and *Research* = “*database*”

Because of the flexibility of natural language, the same meaning can be expressed in different ways. For example, *city of Ottawa* and *Ottawa city* refer to the same thing. To identify categories from the query, we specify the rules for the set of possible strings that we want to match in the regular expressions. For example, there are two regular expressions that we are using: the expression “ $(?: (\backslash S.*) \backslash s+)?(?: \text{cit}(\text{?:y|ies}))\$$ ” can get category *city* in query *Ottawa city*, and “ $(?: \text{cit}(\text{?:y|ies})) \backslash s+(?: \text{of} \backslash s+)?(\backslash S.*)\$$ ” can identify category *city* in query *city of Ottawa*.

Although the regular expression is very useful in the natural language applications ranging from tokenization to parsing [23], due to the ambiguity of natural language [5, 11], there is a limitation in identifying the categories when the names of objects are the same as the topic category. For example, in queries *department of Computer Science* and *department of defence*, *department* is a category of the former query, while *department of defence* is the name of one object in category *Government*. The regular expression cannot handle this case.

To solve this problem, we use the keyword search approach introduced in Subsection 6.3.1. In particular, the keyword search first identifies whether the query is the names of objects. If so, return the results, otherwise, send the query to the natural language parsing.

6.5.2 ORSQL Generation

There are three clauses fundamental to construct ORSQL: SELECT CLAUSE, FROM CLAUSE and WHERE CLAUSE. The ORSQL we translated focuses on these clauses as well.

The FROM CLAUSE may contain several related tables. Actually, this feature of the FROM CLAUSE raises one of the most difficult, and most fundamental problem

in a natural language interface endeavor to relational database [28]. Because the query somewhat does not match the database structure, if two concepts containing in a query are found to be represented respectively by two attributes (say A1 and A2), when A1 and A2 are found in different relations (say R1 and R2), the natural language interface has two choices. It can give up as many other natural language interfaces suggest [28], or it can try to guess how R1 and R2 can be joined together, possibly via other relations. Worse, if there are more than one way of joining together, say R1 join R2 join R3 and R1 join R4 join R2, the natural language interface must take an intelligent choice. We solve this problem as follows. According to the category model, one category contains all the necessary attributes and relationships. The queries based on the natural language patterns focus on the objects in the topic category. After identifying the topic category from the query (see Subsection 6.5.1), we can find the attributes in this topic category rather than determining the possible join path of relations. In addition, since only one category– topic category is involved, the join of relations is avoided. Figure 6.7 shows the relationships among the category, object type, and object table. One category corresponds to one object table. Therefore, the FROM CLAUSE that we generate contains only one table which corresponds to the topic category.

Figure 6.7: Relationships among category, object types, and object table

On the other hand, one object table may store more than one object types which have inheritance relationships. In order to map the topic category to the corresponding object table, we need to map the object type to the object table. For example,

the topic category *department* in the query *Find department of Physics at Carleton University*, has a corresponding object type *department.t*. To generate the FROM clause, the system should know the tuples with object type *department.t* are stored in the object table *academicUnit_tab*. We create a stored procedure, *GetTableName-fromType (type name)* to handle this kind of mapping. It accepts the name of the object type and returns the object table name. This stored procedure use the *data dictionary* to build up the mapping. Therefore, the efficiency of the performance is ensured.

We use *Alias Matching Manager* to get the attribute names of the object types. However, to generate the SELECT CLAUSE, only having the attribute names is not sufficient. The representation of the attributes in ORSQL also depends on the attributes types. We develop a procedure to do this. The appropriated ORSQL representation of the types is as follows:

1. Primary types. Put the tuple variable as the prefix of the attribute name.
2. OID. To represent what the OID references to, we put the tuple variable as the prefix of the names of attributes and indicate what the attributes of the tuple we want.
3. Object types. Put the object type name in front of the attributes name: *type name.attribute name*.
4. Nested table. An additional tuple variable representing the nested tuple should be defined.
5. Varray. The representation is the same as the nested table.

For example, ORSQL for Example 6.5.1 is generated as: *SELECT p.name, p.contact_information.email, p.website FROM professor p*, where *p.name* and *p.website*

are the representation of the primary type and *p.contact.information.email* is the representation of the attribute *email* in object type *contact.information_t*.

Up to now, we have translated two of the *three elements*: the topic category, selected attributes to the FROM CLAUSE and SELECT CLAUSE respectively. The next step is to translate the third element– conditions to the WHERE CLAUSE. Although Pattern 1 allows to express very complex conditions in queries, translating conditions to the WHERE CLAUSE has obvious rules. These conditions can be abstracted as *Noun + relation operator + values*. The *Noun* comes from three kinds of *semantic components*: topic category, the attributes of the topic category and condition categories. The values correspond to these three aspects. The *relation operator* is shown in Table 6.1. Suppose the object table corresponding to the topic category is T_1 ; the value of the topic category is V_1 ; the attributes of the object type corresponding to the topic category are represented as A_1 , A_n , the values of the attributes are V_{a1} V_{an} ; the object tables corresponding to the condition categories are T_{c1} T_{cn} , values are V_{c1} V_{cn} . The translation of conditions to the WHERE CLAUSE is presented as follows.

- If topic category has value, the condition can be translated into ORSQL as:
 $T_1.name = V_1$. The attribute *name* is inherited from the root object *obect_t*, every table has this attribute.
- If conditions are the attributes of the topic category, translate the attributes of the topic category into the appropriate ORSQL representation (see the above discussion about translating select attributes into the SELECT CLAUSE). The formulation is: $T_1.A_1 + relation\ operator + V_{a1} + and/or + T_1.A_2 + relation\ operator + V_{a2} + + T_1.A_n + relation\ operator + V_{an}$. *And* and *or* are generated from Subsection 6.5.1.

- Since we use one table in the FROM CLAUSE, if conditions come from the condition categories, the *object navigation* (see Chapter 2) is used to represent these condition categories corresponding to tables $T_{c1} \dots T_{cn}$ by the attributes of table T_1 . Because condition categories have relationships with the topic category, there must exist at least one attribute A_i in T_1 which is the start point in the path comprised by OID from the topic category to condition categories (see Figure 4.8). Following this path, we can represent the table $T_{ci} \in \{T_{c1} \dots T_{cn}\}$ by $T_1.A_i.A_{ci1} \dots A_{cin}$, $A_{ci}.A_{ci1} \dots A_{cin}$ is the *object navigation* path and A_{cin} references to the object type of T_{ci} . It is possible that we may get more than one path. Although every path can reach the destination, we'll choose the shortest one from the system efficiency point of view.

Values in the topic category, condition categories and whose clause may be mapped to the different data types representation according to the corresponding attribute types. For example, in Example 6.5.1 since the type of attribute *Research* is *String*. The corresponding ORSQL representation is adding quote marks to the value, e.g. 'Database'. If the corresponding attribute type is *Number* and there is a quantifier to express the number, such as 1 million, 2.5 thousand, 0.11 billion, etc. we need translate these quantifiers into number formation such as 1000000, 2500, 11000000. Values with *Number* type needn't add quote marks.

There are two conditions in Example 6.5.1: (1) *University* = 'Carleton University' (2) *and research* = 'Database'. To translate condition (1) to the WHERE CLAUSE, we need use *object navigation*, which uses attributes in table *professor* to represent attribute *Name* in table *university*. According to the path from category *professor* to category *university* shown in Figure 4.8, we represent the condition (1) as: *p.academic_unit.university.name* = 'Carleton University'. "p" is the tuple value of table *professor*. Condition (2) is represented as: *p.research* = 'Database'.

After we get three clauses above, the final ORSQL can be generated by combining them: `SELECT CLAUSE + FROM CLAUSE + WHERE CLAUSE`. The final translated ORSQL for Example 6.5.1 is : *SELECT p.name, p.address.email, p.website FROM professor p where p.academic_unit.university.name = 'Carleton University' and p.research = 'Database'*. Figure 6.8 shows the query results gotten from our system.

There are 3 results were found!

- 1. Leopoldo Bertossi
- 2. Mengchi Liu
- 3. Sivarama Dandamudi

NAME	FNAME	WEBSITE
LNAME	EMAIL	
NAME	Leopoldo Bertossi	
FNAME	Leopoldo	
LNAME	Bertossi	
EMAIL	bertossi@scs.carleton.ca	
WEBSITE	http://www.scs.carleton.ca/~bertossi	

NAME	FNAME	WEBSITE
LNAME	EMAIL	
NAME	Mengchi Liu	
FNAME	Mengchi	
LNAME	Liu	
EMAIL	mengchi@scs.carleton.ca	
WEBSITE	http://www.scs.carleton.ca/~mengchi	

NAME	FNAME	WEBSITE
LNAME	EMAIL	
NAME	Sivarama Dandamudi	
FNAME	Sivarama	
LNAME	Dandamudi	
EMAIL	sivarama@scs.carleton.ca	
WEBSITE	http://www.scs.carleton.ca/~sivarama	

Figure 6.8: Results of the query: Find name, email, website of professors in Carleton University whose research includes database.

6.6 Some Examples

In this section, we provide some examples to demonstrate how to translate a query to ORSQL.

We begin with the query: *find cities in Canada whose population is more than 0.3 million*. This query is based on Pattern 1. It has the *Value without Category* formation, because there is no a category for keyword “Canada” in the query. So, *Keyword Processor* is used to find the categories for keyword “Canada”. Figure 6.9 shows the interface for users to select the related categories for keyword “Canada” when there are same *priority values*. After the category identified, the query is converted as: *find cities in country Canada whose population is more than 0.3 million*

Please select the category you mean for: canada

Figure 6.9: User interface: select related categories

The selected attributes, topic category, and conditions are extracted as follows:

- The selected attributes: null.
- The topic category: City
- Conditions: (1)the condition from condition category: country='Canada', (2)

the condition from the whose clause: $\text{population} > 0.3 \text{ million}$.

The processes of the ORSQL generation is shown as follows:

- The FROM CLAUSE: The topic category *City* corresponds to object type *City.t*. The corresponding object table *City_tab* can be gotten through stored procedure *GetTableNamefromType (type name)*. Therefore, the FROM CLAUSE is: FROM *City_tab*.
- The SELECT CLAUSE: Since the selected attributes is null, the SELECT CLAUSE contains all the attributes. Therefore, it is: SELECT *.
- The WHERE CLAUSE: To translate condition (1), we need represent attribute *Name* in object table *Country_tab* by attribute *Province* in object table *City_tab*, because only object table *City_tab* involves in the FROM CLAUSE. The *object navigation* path from category *City* to *Country* is: City - Province- Country (see Figure 4.8). We need to represent this path as the corresponding ORSQL formation. The condition (2) is translated as: $\text{Population} > 300000.0$. The final ORSQL is: SELECT * FROM CITY_TAB x WHERE (x.POPULATION > 300000.0) AND (x.PROVINCE.COUNTRY.name= 'Canada').

The query results are shown in Figure 6.10. We also can use the with phrase to replace the whose clause as : *find cities in Canada with population over 0.3 million*, or use a query according to the variants of Pattern 1: *Are there any cities in Canada whose population is not less than 3000000 ?*

Query *Find countries whose population is maximal* has the aggregation function.

The selected attributes, topic category, and conditions are extracted as follows:

- The selected attributes: null.

find cities in Canada whose population is more than 0.3 million

There are 4 results were found!

• NAME	• AREA	• POPULATION	• PROVINCEORSTATE	• WEBSITE
1. Calgary				
2. Edmonton				
3. Ottawa				
4. Toronto				

• NAME	• AREA	• POPULATION	• PROVINCEORSTATE	• WEBSITE
NAME			Calgary	
PROVINCEORSTATE			PROVINCEORSTATE	
AREA			0	
POPULATION			864500	
WEBSITE				http://www.gov.calgary.ab.ca

• NAME	• AREA	• MAP_URL	• POPULATION	• PROVINCEORSTATE	• WEBSITE
NAME				Edmonton	
MAP_URL		http://maps.maps.yahoo.com/mapimage?MAPData=3P0h4yhzy3GjIF2TWQcE_Y_d342nznqPSkOmVpxUEQ5QWJjnFSDIcjSMwQMIBow5223BDIIOnGQ3Xusi7tKKE8BcwC8sxx0gLeD2 G9yxhQpqi5fInUCGxu72v_lcu8w-		PROVINCEORSTATE	
PROVINCEORSTATE				PROVINCEORSTATE	
AREA			0		
POPULATION			693700		
WEBSITE					http://www.gov.edmonton.ab.ca

• NAME	• AREA	• MAP_URL	• POPULATION	• PROVINCEORSTATE	• WEBSITE
NAME				Ottawa	
MAP_URL		http://maps.maps.yahoo.com/mapimage?MAPData=a0Eic_hyzy0d5B.kvupY7ak9EfeqU8BqGt9IHRH5BPUBxMx0hzptJgRzqslk_s9JUCy5hk_tB9r7pvcRb92obCQqRWYtUW_XA1O9N3V3WKhNlNqWUKBxkaA		PROVINCEORSTATE	
PROVINCEORSTATE				PROVINCEORSTATE	
AREA			0		
POPULATION			348500		
WEBSITE					http://www.city.ottawa.on.ca

Figure 6.10: Results of the query: Find cities in Canada whose population is more than 0.3 million

- The topic category: Country
- Conditions: the condition from the whose clause: population = maximal

The processes of the ORSQL generation is shown as follows:

- The FROM CLAUSE: The corresponding object table of the topic category *Country* is *Country_tab*. Therefore, the FROM CLAUSE is: FROM City_tab.
- The SELECT CLAUSE: SELECT *.

- The WHERE CLAUSE: the *aggregation keyword*: *Maximal* can be considered as a special value. To get this value we need the following ORSQL: SELECT MAX(c.POPULATION) FROM COUNTRY_TAB C. Therefore, the final ORSQL is: SELECT * FROM COUNTRY_TAB x WHERE (x.POPULATION = (SELECT MAX(c.POPULATION) FROM COUNTRY_TAB C)).

The query results are shown in Figure 6.11.

There are 1 results were found!

1. China		
<ul style="list-style-type: none"> NAME PEOPLE COMMUNICATIONS TRANSNATIONALISSUES 	<ul style="list-style-type: none"> CODES GOVERNMENT TRANSPORTATION 	<ul style="list-style-type: none"> CURRENCY GEOGRAPHY ECONOMY MILITARY
NAME	China	
CODES	NAMEREPRSCODES	
CURRENCY		
	LOCATION	Eastern Asia, bordering the East China Sea, Korea Bay, Yellow Sea, and South China Sea, between North Korea and Vietnam
	GEOCOORDS	35 00 N, 105 00 E
	TOTAL	9596960
	LAND	9326410
	WATER	270550
	TOTAL	22147.24
	LANDBOUNDS	Afghanistan 76 km, Bhutan 470 km, Burma 2,185 km, Hong Kong 30 km, India 3,280 km, Kazakhstan 1,532 km, North Korea 1,416 km, Kyrgyzstan 858 km, Laos 423 km, Macau 0.34 km, Mongolia 4,676.9 km, Nepal 1,236 km, Pakistan 523 km, Russia (northeast) 3,605 km, Russia (northwest) 40 km, Tajikistan 414 km, Vietnam 1,281 km
	COASTLINE	14500
	CLIMATE	extremely diverse; tropical in south to subarctic in north
	TERRAIN	mostly mountains, high plateaus, deserts in west; plains, deltas, and hills in east
GEOGRAPHY	ELEVEXTREMES	LOWESTPOINT Turpan Pendi -154 m HIGHESTPOINT Mount Everest 8,850 m (1999 est.)
	ARABLELAND	10
	PERMCROPS	0
	PERMPASTS	43
	FORESTS	14
	OTHER	33
	IRRILAND	408720
	NATHAZARD	frequent typhoons (about five per year along southern and eastern coasts); damaging floods; tsunamis; earthquakes; droughts

Figure 6.11: Results of the query: Find countries whose population is maximal.

Table 6.2 shows some queries and the corresponding ORSQL.

No.	Queries	ORSQL
1	Find professor White	SELECT * FROM PROFESSOR X WHERE (UPPER(X.NAME) LIKE UPPER('%white%'))
2	Find professor White in Carleton University	SELECT * FROM PROFESSOR X WHERE (UPPER(X.NAME) LIKE UPPER('%White%')) AND (UPPER(X.ACADEMICUNIT.UNIVERSITY.NAME)='Carleton University'))
3	Find professor White in Computer Science Department at Carleton University	SELECT * FROM PROFESSOR X WHERE (UPPER(X.NAME) LIKE UPPER('%White%')) AND (UPPER(X.ACADEMICUNIT.name) LIKE UPPER('%Computer Science%')) AND (UPPER(x.ACADEMICUNIT.UNIVERSITY.name) = UPPER('Carleton University'))
4	Find universities in Ontario	SELECT * FROM UNIVERSITY_TAB X WHERE (UPPER(X.CITY.PROVINCEORSTATE.NAME) LIKE UPPER('%Ontario%'))
5	Find population of Province Ontario	SELECT X.NAME, x.POPULATION FROM PROVINCE_TAB x WHERE (UPPER(x.NAME) = UPPER('Ontario'))
6	Find countries whose population is more than 1 million and GDP is not less than 2000000	SELECT * FROM COUNTRY_TAB X WHERE (X.PEOPLE.POPULATION > 1000000.0) AND (X.ECONOMY.GDP >= 2000000.0)
7	Find universities in Canada whose programs include MBA and graduate tuition is not more than 2120	SELECT * FROM UNIVERSITY_TAB X, TABLE(x.FACULTY.COLLEGES) XXXY, TABLE(VALUE(XXXY).ACADEMICUNIT) XXXYY, TABLE(VALUE(XXXYY).PROGRAMS) XXXYYY WHERE (UPPER(VALUE(XXXYYY).PROGRAM) LIKE UPPER('%MBA%')) AND (X.FEES.FEEYEAR < 2120) AND (UPPER(X.ADDRESS.COUNTRY) LIKE UPPER('%Canada%'))
8	How many departments in Carleton University	SELECT COUNT(*) FROM ACADEMICUNIT_TAB X WHERE (UPPER(X.UNIVERSITY.NAME)='Carleton University'))
9	Find countries whose GDP is over average	SELECT * FROM COUNTRY_TAB Y WHERE Y.ECONOMY.GDP >= (SELECT AVG(X.ECONOMY.GDP) FROM COUNTRY_TAB X)

Table 6.2: Queries and ORSQL

6.7 User Interface

As mentioned early, we use JSP (Java Server Page) to develop a web browser based user interface which can be deployed into either Internet or Intranet. Browser based interface is portable, because users only use browsers to communicate with system without any other additional resource requirements, for example client programs. The goal of developing this user interface attempts to provide an easy integrated environment for users get the exact information they want. Figure 6.12 shows the home page of the system. There is one input field for users to enter their queries. The search button submits users' requests to the system. The *categories search* provides users with a hierarchy of categories, so that users can simply click on some categories to move down the hierarchy and find what they want.

It is important that the user interface is built independently from data resources. Therefore, the interface is *stable* even when the data or data structure changes over times. This user interface reflects the properties of the category model by handling all kinds of data types in the model. For example, we use hyperlinks to represent the relationships of the categories, so that not only can users find the information of one object but also are able to navigate related objects in different categories by clicking these hyperlinks. Figure 6.13, 6.14 and 6.15 show the processes of navigating from *country Canada* to one of its provinces *Ontario*.

To maintain natural relationships between data, we fully utilize the database design rather than link these content manually. Thus, this interface makes the data organization of the category model visual to users.

Figure 6.12: User interface

NAME	Canada	
CODES	GOVERDATACODE	CA
	NAMEREPRESCODES	TWOCHARALPHA CA THREECHARALPHA CAN THREENUM 124
	TELCODE	1
	INTERNETCODE	.ca
MAP_URL	http://www.cia.gov/cia/publications/factbook/maps/ca-map.jpg	
FLAG_URL	http://www.cia.gov/cia/publications/factbook/flags/ca-flag.jpg	
CAPITAL	CAPITAL	
CURRENCY	CURNAME	Canada dollar
	ABBREV	CAD
	SUBDIVISION	1 Can\$ = 100 cents
PROVINCESORSTATES	EXCHRATE_URL	http://www.eanda.com/convert/classic
	PROVINCESORSTATES	

Hyper link to Canadian provinces

Figure 6.13: Country Canada

1	Yukon Territories
2	Northwest Territories
3	Saskatchewan
4	Quebec
5	Prince Edward Island
6	Ontario
7	Nova Scotia
8	Newfoundland
9	New Brunswick
10	Manitoba
11	British Columbia
12	Alberta
13	Nunavut

Links to
gov here
Before

Figure 6.14: Provinces in Canada

NAME	Ontario
COUNTRY	COUNTRY
ABBREV	Ont.
POSTABBREV	ON
	<ul style="list-style-type: none"> • London 519 • North Bay 705 • Ottawa 613
TELCODES	<ul style="list-style-type: none"> • Thunder Bay 807 • Toronto Metro 416 • Toronto Vicinity 905
MAP_URL	http://img1.maps.yahoo.com/mapimage?MAPData=kFgGjvhyzyOP73D&oh8Xu13xUYRYM_BKT8pA_6hOJFbILc4NCB8CGrUqnp.yDhg0uy5eymIkvuAyUBk0EQNvHWTC6kidn9EEQUE7hikDILL_YOQRORBTWtabrEZqF6v
CAPITAL	CAPITAL
AREA	1068582
POPULATION	12047400
LARGESTCITY	LARGESTCITY
CITIES	CITIES
WEBSITE	http://www.gov.on.ca

Figure 6.15: Province Ontario

Chapter 7

Conclusion

In order to solve the difficulty of finding exact information under this information overload era, this thesis presents the design of a conceptual model for meaningful semi-structure or unstructured data modeling and the natural language interface based on the category model for searching data in a semantic way. We have also developed a prototype system to prove the feasibility of our approaches.

The category model is critical. It is defined based on the natural view that the real world entities can be classified by categories. The notion of *category* can not only be used to organize objects meaningfully but also help maintain the semantic meanings of objects. The category hierarchy and category relationships make categories naturally interconnected. Therefore, through the connection, when objects in one category are identified, the objects in the related categories also can be reached. To support the semantic integrity of relationships, the *dependency relationship* is introduced in the category model. To naturally model real word entities, the category model removes the restrictions of normalization and provides the complex value types such as OID, tuple, and set. The category model is defined independently from the data resources. Thus, it is stable even when the data resources change over times. The category model

diagram is presented in this thesis as a system analysis tool spanning the development cycle.

Object-relational databases are the latest generation of the database after relational databases, and object-oriented databases. It is based on the relational database, adding the object-oriented features, such as object identifiers, inheritance, complex object types, etc. The features of object-relational databases technically ensure the integrity of the migration from the category model to the database schema. In other words, all elements in the category model have the explicit representations in the object-relational database. So, the object-relational database schema inherits the semantic meanings of the category model. This character is important, not only because it makes the database design easy but also because it simplifies the natural language parsing action.

The natural language interface provides an effective channel for users to express their queries in a semantic way. The successful development of natural language interface to databases will greatly expand database applications. Our natural language interface approaches fully utilize the semantic meanings, captured in the category model, to distinguish the ambiguity of natural languages rather than depend on the grammar or rules. In addition, the natural language interface built based on this approach overcomes the portability problems caused by the limitation of the knowledge domain representation.

In order to bridge the gap between users' query needs and the implementation of the system, this thesis defines several natural language patterns which utilize the category model's fundamental concepts such as category and category relationships. The function of these patterns is to query objects in a topic category and the constraint conditions may come from different categories. Utilizing these patterns, users can assemble comparison conditions (e.g. population is more than 300 million) or

Boolean conditions. At the same time, there are several patterns providing the aggregation functions such as *minimal*, *maximal*, *average*, *count*, and *sum*. Thus, these patterns have the capability of constructing complex queries to meet users' needs.

This thesis presents the implementation of the prototype system. In order to translate the query into ORSQL, there are two processors. The *keyword processor* converts the query into *Category + Value* formation, which is required by the next parsing steps. Another processing is the *natural language processor* which includes two procedures: *alias matching manager* and *natural language parser*. Alias matching manager solves the problem of mapping synonyms to the database schema. Natural language parser translates the query refined by previous procedures to ORSQL. It is important to note that the translation processes fully utilize object-relational database features, such as the *object navigation* which allows attributes of one table being represented by other table's attributes. In other words, the techniques of object-relational databases ensure the feasibility of the translation processing.

In the prototype system, the category model organizes the semi-structure or unstructured data to the high structured information. Benefited from this high structured information, the system can do the comparison, calculation and inference based on the contents rather than the syntactically retrieval. This prototype system shows that if we can model the data by their semantic meaning and relationships, then a natural language interface is a reachable objective.

The methodologies of the category model and natural language interface presented in this thesis have the wide applications, for example web search engines, the information systems, the knowledge management systems, e-business, etc.

7.1 Future Work

The future research can be described from two aspects: the multiple classifications in the category model and extension to the natural language patterns.

Multiple classification means that one object may be classified into different categories, and at the same time may change its categories during the course of its lifetime. This problem is mentioned as *object evolution* in [16, 27, 32, 33, 34, 35]. Multiple classification is a basic form of evolution and is not yet supported in the existing object-relational database management system. The future research will focus on the representation of multiple classification in the category model, and the corresponding implementation in the object-relational database.

Because of the flexible expression of natural language, we would like to extend the quasi-natural language into a full-fledged one by extending patterns and adding other useful features to allow more complex queries. Our objective is to build the natural language patterns that have the same expressive power as ORSQL.

Bibliography

- [1] A. Albano and R. Bergamini. An object data model with roles. In *19th International Conference on very large databases , Proceedings*, pages 39–51. Dublin, Ireland, 1993.
- [2] I. Androustopoulos, G.D. Ritchie, and P. Thanisch. Natural language interfaces to databases—an introduction. *Journal of Language Engineering*, 1(1):29–81, 1995.
- [3] T. Berners-Lee. *Weaving the Web*. Harpur, San Francisco, 1999.
- [4] E. Bertino and G. Guerrini. Extending the ODMG object model with composite objects. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems*, pages 259–270, 1998.
- [5] E. Brill. Pattern based disambiguation for natural language processing. In *Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, pages 141–148, 2000.
- [6] P. Brown. *Object-Relational DBMS*. Morgan Kaufmann Publisher Inc, 2 edition, 1999.
- [7] R. Catell. *Object data managemnet*. Addison-Wesley, 1 edition, 1994.

- [8] R. G. G. Cattell and D. Barry, editors. *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann, 1999.
- [9] S. K. Cha and Gio Wiederhold. Kaleidoscope data model for an english-like query language. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 351–361. Morgan Kaufmann, 1991.
- [10] P. Chen. The entity relationship model: Toward a unified view of data. In *ACM Transaction on DB system 1.1*, pages 9–36, 1976.
- [11] W. W. Chu and F. Meng. Database query formation from natural language using semantic modeling and statistical keyword meaning disambiguation. Technical Report 990003, 16, 1999.
- [12] S. Decker, D. Fensel, F. v. Harmelen, I. Horrocks, S. Melnik, M. C. A. Klein, and J. Broekstra. Knowledge representation on the web. In *Description Logics*, pages 89–97, 2000.
- [13] S. Decker, S. Melnik, F. v. Harmelen, D. Fensel, M. C. A. Klein, J. Broekstra, M. Erdmann, and I. Horrocks. The semantic web: The roles of XML and RDF. *IEEE Internet Computing*, 4(5):63–74, 2000.
- [14] D.W. Embley, B.D. Kurtz, and S.N. Woodfield. *Object-Oriented Systems Analysis: A Model-Driven Approach*. Yourdon Press, Prentice Hall, 1992.
- [15] J. Friedl. *Mastering regular expressions*. O'Reilly&Associates, Inc, 2 edition, 2002.
- [16] G. Gottlob, M. Schrefl, and B. Rock. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3):268–296, 1996.

- [17] T. Griffiths, J. McKirdy, G. Forrester, N. W. Paton, J. B. Kennedy, Peter J. Barclay, R. Cooper, A. Carole, and P. D. Gray. Exploiting model-based techniques for user interfaces to databases. In *VDB*, pages 21–46, 1998.
- [18] M. D. Guarino. Formal ontology and information system. In *FOIS 98, Trento, IOS Press*,, pages 3–15, 1998.
- [19] G. Hendrix, E.sacerdoti, D.sagalowicz, and J.slocum. Developing a Natural Language Interface to Complex Data. In *ACM Transactions on Database Systems*, pages 3(2):105–147, 1978.
- [20] I. Jacobson. *Object-Oriented Software Engineering, A use Case Drivern Approach*. Addison-Wesley, 1992.
- [21] T. Johnson. *Natural Language Computing: The Commercial Applications*. Ovum Ltd. London, 1985.
- [22] J. L. Johson. *Database, Models, Languages, Design*. Oxford University Press, 1997.
- [23] L. Karttunen, J. Chanod, G. Grefenstette, and A. Schiller. Regular expressions for language engineering, 1996.
- [24] S. Khoshafian and G. Copeland. Object identity. In *OOPSLA'86 as ACM SIGPLAN 21*, pages 406–416, 11 1986.
- [25] W. KimA. Object-oriented database: Definition and research direction. In *IEEE Transactions on Knoledge and Data Engineering*, pages 2(3):317–341, 1990.
- [26] Kuhn and R.De Mori. The application of semantic classification trees to natural language understanding. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 17(5):449–460, 1995.

- [27] J. Lagorce, A. Stockus, and E. Waller. Object-oriented database evolution. In *ICDT*, pages 379–393, 1997.
- [28] W. S. Luk. Building natural language interface to an er database. In Frederick H. Lochovsky, editor, *Entity-Relationship Approach to Database Design and Querying, Proceedings of the Eight International Conference on Entity-Relationship Approach, Toronto, Canada, 18-20 October, 1989*, pages 345–360. North-Holland, 1989.
- [29] X. Meng and S. Wang. Nchiql: The chinese natural language interface to databases. In Heinrich C. Mayr, Jirí Lazanský, Gerald Quirchmayr, and Pavel Vogel, editors, *Database and Expert Systems Applications, 12th International Conference, DEXA 2001 Munich, Germany, September 3-5, 2001, Proceedings*, volume 2113, pages 345–360. Springer, 2001.
- [30] S. Miller, D. Stallard, R. Bobrow, and R. Schwartz. A fully statistical approach to natural language interface. In *The 34th annual meeting of the association for computational linguistics*, pages 55–61. Santa Cruz, CA, 1996.
- [31] B. Mitchell. *Theory of categories*. Academic Press INC London LTD., 2 edition, 1967.
- [32] M. Norrie. *Distinguishing Typing and Classification in Object Data Models*. In *Information Modelling and Knowledge Bases*, chapter 25. IOS, 1995., 1995.
- [33] M. C. Norrie. An extended entity-relationship approach to data management in object-oriented systems. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 390–401, 1993.

- [34] M. C. Norrie, A. Steiner, A. Wurgler, and M. Wunderli. A model for classification structures with evolution control. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 456–471, 1996.
- [35] A. Olivé, D. Costal, and M. Sancho. Entity evolution in isa hierarchies. In *Conceptual Modeling - ER '99, 18th International Conference on Conceptual Modeling, Paris, France, November, 15-18, 1999, Proceedings*, volume 1728 of *Lecture Notes in Computer Science*, pages 62–80. Springer, 1999.
- [36] OMG. Unified modeling language specification version 1.5, <http://www.omg.org/technology/documents/formal/uml.htm>.
- [37] S. Navathe R. Elzmasri. *Fundamentals of Database System*. Addison-Wesley, 3 edition, 2000.
- [38] D. Rudloff, F. Beuvron, and M. Schlick. Extending tableaux calculus with limited regular expression for role path: an application to natural language processing. In *Description Logics, Proceedings of the 1998 International Workshop on Description*, 1998.
- [39] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modelling and Design*. Prentice Hall International, 1991.
- [40] R. Schank and C. Riesbeck. *Inside computer understanding*. Erlbaum, Hillsdale, NJ., 1981.
- [41] S. Shlaer and S.J. Mellor. *Object Oriented Modelling Systems Analysis: Modelling the World in Data*. Prentice Hall, 1988.
- [42] H.E. Shrobe, editor. *Natural Language Interfaces*. Morgan Kaufmann, 1988.

- [43] H. Stuckenschmidt, F. van Harmelen, D. Fensel, M. Klein, and I. Horrocks. Catalogue integration: A case study in ontologybased semantic translation, 2000.
- [44] A. Cynthia Thompson and J. M. Raymond. Automatic construction of semantic lexicons for learning natural language interfaces. In *AAAI/IAAI*, pages 487–493, 1999.
- [45] G. Vossen. *Data models, database languages and database management system*. Addison-Wesley, 1 edition, 1991.
- [46] T. Winograd. *Understanding Natural Language*. Academic Press, New York, 1972.
- [47] W. A. Woods. *Semantics and quantification in natural language question answering*. Morgan Kaufmann Publishers, Inc. Los Altos, California, 1986.
- [48] W. A. Woods. What’s in a link: Foundation for semantic network. In *Journal of Documentation*, pages 49:188–207, 1993.
- [49] M. Zelle and R. J. Mooney. Learning to parse database queries using inductive logic programming. In *The Thirteenth National Conference on Artificial Intelligence*, pages 1050–1055, Portland, 8 1996.