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.

ORDERS: an Object Relational Database Query Reformulation System

By

Lingyan Xu

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

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

June 2, 2003

© Copyright
2003, Lingyan Xu

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: 0-612-88788-X

Our file Notre référence

ISBN: 0-612-88788-X

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

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis,

ORDERS: an Object Relational Database Query Reformulation
System

submitted by

Lingyan Xu

Dr. Frank Dehne
(Director, School of Computer Science)

Dr. Mengchi Liu
(Thesis Supervisor)

Carleton University
June 2, 2002

Abstract

Object-relational databases have been widely used to store complex objects. Effective information retrieval from object-relational databases with complex database structures is known to be a non-trivial problem. In particular the formulation of valid object-relational SQL queries is a critical step for effective information access in such contexts. Existing approaches usually require more or less extensive knowledge about database management systems, query languages, and the underlying database structures. In this thesis we present ORDERS, an easy-to-use Object-Relational Database Query Reformulation System that allows users to issue queries in their own vocabulary without extensive knowledge of the database management, query languages and the underlying database structures, and automatically generates object-relational SQL statements for execution.

In this system, we use an ontology-based thesaurus to associate application level concepts with database schema objects. Based on this association, a fuzzy matching mechanism is used to map user concepts to database concepts.

As users do not have extensive knowledge about the database structures, user queries may not specify all relevant schema objects. The system maintains a schema graph that represents the database structure based on the database schema, uses it as a road map to locate proper schema objects and find optimal query paths for final object-relational SQL statements, using the breadth-first search algorithm.

Acknowledgments

First of all, I would like to thank Professor Mengchi Liu, my supervisor, for his excellent guidance, encouragement, responsiveness, and financial support through my research. His high academic standards have made the learning process a valuable experience that will benefit me the rest of my life. I also appreciate the TA offer from the School of Computer Science.

I want also to thank Mr. Wenjie Tong, Mr. Xiaofei Jia and Mr. Tao Fu for their kindly and generous help.

I am grateful to my family for their consistent support and understanding. Thanks to my parents Zhengxian Zhang and Duanyi Xu, for their encouragement and support all through my life. Special thanks to my husband Wenlu Wang, and to my daughter Janet, for their love and supports all the way long.

Contents

Abstract	iii
Acknowledgments	iv
List of Tables	vii
List of Figures	viii
1 Introduction	1
1.1 Motivation	1
1.2 Problem Statement	3
1.3 Our Approach	4
1.4 Thesis Outline	7
2 Related Works	8
2.1 Database Queries at the Logical Level	9
2.2 Conceptual Queries	11
2.3 Data Access Techniques at the External Level	12
3 Object-relational Technology	15
3.1 Why ORDBMS?	15
3.2 SQL:1999 Standard and Oracle9i	17

3.2.1	New OO features supported by SQL:1999 and Oracle9i	18
3.2.2	More about Oracle9i	23
4	Overview of ORDERS	24
4.1	System Architecture	24
4.2	A sample application	27
5	Schema Graph	31
5.1	Schema Graph	31
5.1.1	Concept of the schema graph	32
5.1.2	Example schema and schema graph diagrams	34
5.2	Thesaurus and Fuzzy Matching Mechanism	40
5.3	Schema Graph Implementation and Generation of Schema Graph	45
5.4	Properties of schema graph	49
6	ORSQL Query Reformulation	50
6.1	Generation of query trees	50
6.1.1	Query path and query tree	50
6.1.2	Construction of Query Trees	53
6.1.3	Handle of Selection Predicates	58
6.1.4	Optimization in query trees	59
6.2	Generation of ORSQL queries	61
6.3	Query Examples	65
7	Conclusion	71
7.1	Summary	71
7.2	Future work	74
	Bibliography	76

List of Tables

6.1	ORDERS datatypes, SQL:1999, and Oracle9i datatype mapping . . .	63
-----	---	----

List of Figures

3.1	Graphical view over the relationships and structure of objects for the University-Department-Professor example	21
4.1	ORDERS architecture	25
4.2	User Interface of the Intelligent Information System	28
4.3	Architecture of the Intelligent Information System	29
5.1	Schema of the City-University database	35
5.2	Schema graph of the City-University database	36
5.3	Schema for the University-Employee database	37
5.4	Schema graph for the University-Employee database	38
5.5	Query paths for the query “list the name of instructors in Carleton University”	44
5.6	Structure representation of the schema graph for the Professor.tab and its type defined in Figure 5.1	48
6.1	Query tree for the query: name all the professors at Carleton University	57

Chapter 1

Introduction

1.1 Motivation

As computational power and storage capacity increase, accessing and processing large data sets with complex data and relationships play an increasingly important role in many scientific research domains [1, 31, 43] and web-based database applications. With the support of object-oriented features, such as object references and structured data types, object-relational data model has been widely used to mimic real world objects and the relationships among them. A remarkable example arises in querying restructured information from the World Wide Web. Modeling and organizing the semi-structured web data using object-relational database technologies provides the ability to issue structured queries against the database in a fashion that is absent from keyword search methodologies [18, 30]. The process of retrieving information from the databases however, is complex and daunting for non-expert database users.

Existing data retrieval systems almost non-exclusively require the use of query languages that demands more or less extensive knowledge of Database Management

Systems (DBMS), query language syntax, and database structures. As one of the most commonly used database query languages, SQL has relatively simple English-like syntactic and semantic structures. However, it is mandatory to have a thorough knowledge about the database schema and SQL syntax in order to formulate valid database queries. Complicated and advanced informational needs encountered in science, business, and office automation demand automatic and semantic query methods in database applications.

Efforts have been made by database application developers and DBMS vendors to provide user-customized information selection methods. Solutions such as query-by-form [19] and Query by Example (QBE) [45] provide interfaces that display the underlying schema via forms or template lists. Conceptual query languages [12] provide interfaces to guide users through the query formulation process by display the conceptual structure of the database. By using these methods, users can navigate the database structure and specify their queries by selecting and filling in conditions for attributes of desired objects, and are freed from having to memorize the database schema or learning about the rigid query language syntax rules. The major problem of these approaches however, is the limitation of scalability. When the structure of the underlying database gets considerably complex, it is impossible to show the whole database structure in a single shot, and consequently causes difficulties for users to browse through the schema structure screen by screen, and to manually select the desired objects. Furthermore, without a clear picture of how the objects related to each other in the database, it would be difficult for users to find the correct and effective access path from one relation to the other through blind navigation.

1.2 Problem Statement

The objective of this thesis is to design and develop an easy-to-use object-relational database query reformulation system that allows users to issue queries in their own vocabulary without extensive knowledge of the database management, query languages and the underlying database structures, and automatically generates ORSQL statements. To achieve this, there are two major problems to be addressed. The first one is how to match the key terms represented in the user queries into concepts defined in the database schema. The second problem is how to generate efficient query paths that specify what schema structures to be involved in the query and the sequence of accessing them, and to formulate valid SQL queries based on the query paths and the SQL syntax.

In response to the demand of representing object in databases, object-relational management systems are introduced to extend the traditional relational model with new OO features, such as user-defined data types, object identifiers, and object inheritance, etc. [29, 42]. By using the object-relational technologies, database designers are able to model real world objects in a concrete way. However, it is often difficult to map key terms from user queries into database concepts directly, because users usually use a richer vocabulary in their expressions. Without context, semantic vagueness in user queries often causes problem in the determination of the accurate meaning of the terms used in their queries.

Moreover, without extensive knowledge about the database structures, users may not be able to specify all relevant schema objects in their queries. To automatically generate the query paths that construct the frame of the destination SQL queries become a necessary requirement for our system. However, the objects in the real world are related directly or indirectly, the database schema inevitably contains structures

of self-references or circular references. To choose effective and correct query paths over such complex databases becomes a nontrivial task. The selection of query paths also determines the ORSQL operations used in the query, which yield different performance during the query execution. In order to minimize the execution cost, the optimal query generation becomes an important part in ORDERS as well.

1.3 Our Approach

In this thesis we introduce a prototype and the experimental implementation of an Object Relational Database quERY Reformulation System (ORDERS), as a heuristic solution to semantically map the key words in user queries into schema objects, and automatically generate ORSQL queries.

When we issue a request to a database, generally there are three components presented in the query: 1) the subject, the category or the type of the “thing” of interests; 2) what we want to know about the subject; 3) with what condition. For example, consider the query: *find the phone number and address of the professor whose name is Smith*. In this query, professor is the subject or the category, the phone number and address are the attributes we want to know about the professor, and the condition is the name Smith. The basic idea of our approach is to take the three query components as the input and map them into the corresponding elements represented in the SELECT-FROM-WHERE clauses of the ORSQL statement through the following mappings: the attributes of interests are mapped into the attribute list in the SELECT clause, the condition is interpreted into the WHERE clause, and the category (subject) is mapped to the first table in the FROM clause. During this mapping process, the semantic meaning of the user request is concerned.

The reformulation of user queries into valid ORSQL statements has been considered as a nontrivial procedure. The construction of correct and effective ORSQL expressions always involves several complicated processes, such as the identification of relevant tables and attributes, the selection of proper query access paths, and the generation of ORSQL statements based on the syntactic constraint defined in ORSQL. The following sequential steps are involved in the process of generating ORSQL statements from user requests:

1. Extract the semantic concepts from the user query and match them into relevant types, tables, attributes, and other data properties represented in the database.
2. Determine a set of query access paths that specify the relevant schema objects and the sequence to access them.
3. Reformulate the query into an ORSQL statement based on the query access paths and the ORSQL syntax.

At the design stage of a database, the requirements from the prospective database users are usually carefully considered so that the conceptual structure of the database can concisely represent the data domain with a full respect to the users' view [16]. Therefore, users and the database schema should always share a common structural view over the application domain of the system, except that the users use a richer vocabulary to describe the concepts. To bridge the gap between the users and the database, we have established a thesaurus based on a domain-dependant ontology. An ontology is a collection of concepts and their interrelationships that can collectively provide an abstract view over an application domain [13, 23]. In this ontology-based thesaurus, a list of related and analogous words of the concepts from the ontology is maintained and is associated with the equivalent database schema objects. A fuzzy matching mechanism is developed to determine the relevance between the key terms

from user queries and the tables and/or attributes from the database schema. A schema graph model is established to represent the structure of the database and is used to dynamically search for the related schema objects. The breadth-first search algorithm is used to generate the query access paths by traversing the schema graph. An ORSQL formulator is developed to generate ORSQL statements based on the query paths and ORSQL syntax.

The ORDERS is an experimental implementation of the system model that can be attached to a complex object relational database and dynamically formulate ORSQL statements based on the semantic meaning of user queries. One of the possible applications of ORDERS is to be used as a back-end interface to intelligent information retrieval systems that are based on object-relational data models, such as search engines for web data, automatic information systems for large and complex scientific databases.

The main contributions of this thesis are:

1. A heuristic prototype for semantic query reformulation systems is proposed and implemented for object-relational databases.
2. A schema graph is introduced as a solution to represent the database structure as a virtual road map that facilitates the process of searching for relevant tables and attributes from the database. The features of the schema graph include:
 - The schema graph is automatically constructed from the schema rather than being preprogrammed. When the underlying database structure changes, the schema graph can be re-established by restarting the system, so that there is no need to reprogram the whole system.
 - The schema graph provides a concrete representation of the database structure and is used as a road map to locate the relevant schema objects and

the paths to access them.

- The schema graph also records all the necessary meta-data needed in the ORSQL query generation process, consequently eliminates the needs of communicating with the database server, and hence speeds up the user query processing.
3. The fuzzy matching mechanism with the ontology based thesaurus is proposed as a novel approach to associate the application level concepts with the database schema objects, which enables users to issue queries using terms they are familiar with.
 4. Shortest path algorithm is used to automatically find the query paths that construct the frame of the destination ORSQL queries, and to achieve the optimal query generation. So there is no need for users to specify the paths manually.
 5. The design and implementation of the ORSQL query generator provide a solution towards automatic generation of ORSQL queries.

1.4 Thesis Outline

The remainder of this thesis is organized as follows: Chapter 2 reviews the related works. Chapter 3 introduces some of the new features of ORDBMS and the Oracle9i system. Chapter 4 gives a brief overview to the ORDERS system. In Chapter 5, the schema graph model and the data structure are introduced. Chapter 6 describes the ORSQL query reformulation process and algorithm. Some sample queries are also examined. Finally, in Chapter 7 we present the conclusions and the plans for future work.

Chapter 2

Related Works

Since Bachman designed the first generalized DBMS in the early 60's [16], database systems have gone through dramatic evolutions. Efforts have been made to provide user-friendly interfaces towards easier database query processes. There are four main levels at which human may interact with a database system [25]:

- External
- Conceptual
- Logical
- Physical

Starting from the bottom, the physical level is closely tight to the chosen DBMS. All details about the physical storage and access structures, such as the index of stored records, the various storage structures, whether to use pointer chains or hashing, etc., are defined at this level. It is rare for end users to communicate with database system at this level.

The logical level expresses the conceptual schema in terms of data structures and operations supported in the chosen data model (relational, hierarchic etc.). This level always involves communication with the database using query languages, such as SQL, OQL, etc.

At the conceptual level, the information is expressed in its conceptual form. Conceptual query languages have been introduced to compose queries to the database.

At the external level, the actual interfaces and input/output representations are used to work directly with the system, for example through screen forms or printed reports, etc.

Based on these four levels, the following sections give a review over the related works for the currently existing database query assistant methods. Since the physical level often deals with the specific DBMS and the internal storage details, and end users rarely access data directly at this level, we will focus on the other three higher levels, namely the logical level, the conceptual level, and the external level.

2.1 Database Queries at the Logical Level

At the Logical Level, databases are represented and implemented based on the underlying data model, such as relational or object-oriented model. The information is expressed using the logical constructs represented in the data model, for example, by means of tables and keys in the relational model. Logical query languages are used in the process of data retrieval and data manipulation [16, 25].

Among the data models that developed in the '70s, namely the relational model, the network model, and the hierarchical models, the relational model has been considered as the most widely used one in the commercial DBMSs. SQL, QUEL, and

QBE are the three relational database query languages that have been implemented and supported by commercial products.

SQL and QUEL are two database query languages that are implemented as interfaces to several commercial DBMSs. Compared with the operations defined in the relational algebra and the relational calculus which are fundamental for manipulating a relational database, SQL and QUEL are syntactically more similar to English and more expressive. However, people still need to get trained before writing any valid database queries in these languages. Moreover, the full knowledge about the database schema is required in order to prepare an efficient query.

QBE (Query By Example) [45] is a user-friendly relational query language that was developed at IBM research institutes. It differs from the SQL and QUEL in that users do not have to specify a structured query explicitly. Rather, they formulate queries by filling in templates of relations that are displayed on a terminal screen. By showing a list of all relation names on the screen, QBE first allows the users to choose the tables or relations needed to formulate a query. After the templates for the chosen relations are displayed, users move to proper columns in the templates and specify the queries or define the conditions using special function keys. In this way, users do not have to follow any rigid syntax rules for query specification, but to construct examples related to the request by entering variables in the column, and thus are freed from having to structure the whole schema in mind or even to remember the table or attribute names. However, it is often tedious to browse the templates from one to another. For complex queries that involve several relations, one needs to choose the table by hand and specifies the join variables in the columns of templates.

With the emerging of the object-oriented database generation, DBMSs have evolved to support complex data types and decomposable data structures. The integration

of programming languages and database systems gives application developers more power and freedom to model the information domain in object-oriented manners. However, the extended SQL (SQL3) for ORDBMS and the OQL for Object-oriented DBMS often lead to greater complexity in query formulation.

2.2 Conceptual Queries

A conceptual schema expresses the structure of an application domain from the perspective of a human rather than a machine. It facilitates communication between modeler and subject matter experts during the modeling process. Given advantage from the tight coupling between the conceptual schema and the database schema, many conceptual query languages have been proposed to allow users to formulate queries directly on the conceptual schema itself [11, 12]. However, most of these language proposals are academic research topics, with at best prototype tool support. One commercial tool, English Wizard, provides some ability for users to enter queries directly in English, but the tool currently suffers from problems with ambiguity, as well as the correctness of its SQL generation [25].

A typical solution adopted by most conceptual query language tools to represent the database to end users is using attributes, which may cause instability since attributes often evolve into entities or relationships as the application model evolves. Efforts have been made by introducing new query languages based on Object Role Modeling (ORM) or Object-oriented System Modeling (OSM), which pictures the application domain in terms of objects that play roles, thus avoiding the notion of attribute. Some of the most significant ORM-based or OSM-based query languages are RIDL [32], LISA-D [27], and OSM-QL [17]. Although they are considered as very powerful and expressive, their usage is limited due to either the lack of tool support

or the complexity in the user interface [25].

The most recently developed ORM based query language is ConQuer [12]. Compared with other conceptual query languages, ConQuer is claimed as more expressive, easier for novice users, and is supported by commercial tool that transforms conceptual queries into SQL queries. A three-pane interface is provided by ConQuer to guide the user throughout the query construction. Objects stored in the database and their roles are displayed on the screen. Users are allowed to choose the objects and the roles of interest, and click on the roles to explore the criteria that can be applied to the object. Typical queries can be constructed by clicking on objects with the mouse and adding conditions. Users are not required to be familiar with the conceptual schema or table structures in the query formulation process. However, when the database becomes complex, users can still easily get lost by having to explore objects and their roles manually.

2.3 Data Access Techniques at the External Level

At the external level, database query tools are often involved in the communications between users and the information systems. Query by form is a traditional solution that allows users to issue queries directly on a screen form. The form usually reflects the underlying database schema with fields that can be filled in by users with appropriate values or conditions. This form-based interface is well suited to simple queries without complex operations involved, and the scope of the query is visible on a single screen. However, this solution does not work well to express complicated queries. Moreover, users will suffer the same pain as in QBE by that they have to explore the form-based schema manually and get lost when the data structure is considerably complex.

UniGuide [19, 18] is an example of form-based database query search engine developed by Enguix et. al. It stores the “Australian Universities” data in an object-relational database. Two sets of user interfaces are provided separately for simple queries that involve one entity or one set, and for complex queries that have more than one set or entities involved. For a complex search, the user needs to specify the select list, where clause, and other database query operations listed on the advanced query form. For a novice end-user, the interface itself becomes a technical challenge since the interface uses technically termed buttons such as “Group by” or “Order by” to represent the query key words in SQL. In order to understand the form and how to use it, at least some basic knowledge about SQL is required.

Dynamic Query [2, 3, 39, 44] is a recently developed database access method that provides continuous feedback to the user during the query formulation process. Dynamic Query is designed as a mechanism for visualizing multidimensional data with graphical interfaces. The systems present all available options to the users by the means of lists, sliders, buttons and other graphical widgets that allows individual adjustment of each value. Using this method, users can specify queries by using the graphical widgets. The result displayed is then dynamically updated with regard to the values selected by users.

This approach is straightforward and well suited as long as the number of dimensions is fairly small. However, as every new dimension or new data property requires new interface components, this approach does not scale very well. In fact, the frequently high number of dimensions easily results in a complex and confusing interface. Such interfaces might give experienced users no problems, but novice users are likely to suffer. Another problem with this approach is that it cannot be applied to large databases with sufficient performance.

In later papers [14, 34], a two-phase approach to dynamic queries was proposed.

The first phase, which is called *query preview*, allows users to formulate an initial query by selecting desired property values while simultaneously being shown the volume of matching data. This is made possible by the pre-generation of a *volume preview table* that indicates the number of data sets for each property value and intersections. When the result is narrowed to a sufficient size, the *query refinement* phase starts. In this phase actual metadata is collected from the database to be used in a dynamic query interface. Using this two-phase approach, the performance problem regarding larger databases is reduced.

A similar approach called *continuous querying* has been described by Shafer and Agrawal [4]. In their paper they present Eureka, a web-based database exploration search engine. In this approach, after an initial search, users are allowed to specify predicates on different attributes and immediately have the result updated.

The dynamic query approach however, has been done based on a critical assumption that the initial query issued by the user retrieves a result set that contains the target information. Usually the initial query is composed of a single property that is chosen from the data model. If this selected property is not what is considered in the user's mind, the desired objects would not present in the result set, then all the attempts of trying to reduce the results will end up with nothing.

To overcome the shortcomings presented in the above approaches, we have designed the ORDERS as a middle-tier software that can be added to the external interface between the users and the underlying ORDBMS. This system accepts user queries defined in terms from the application domain, automatically locates proper database schema objects, and finally generates SQL queries.

Before we get into details about our database query reformulation system, let us first take a look at the object-relational database technology.

Chapter 3

Object-relational Technology

As introduced in Chapter 1, we use Oracle9i – an ORDBMS to serve as a building block in our system. In this chapter, we first give some background of the object-relational technology, and then introduce some related new features provided by SQL:1999 [15] – the new standardization of ORDBMS technology – and by the Oracle9i [21, 28] database management system.

3.1 Why ORDBMS?

The success of Relational DBMSs in the past decades is well known. However, the traditional relational data model and earlier version of SQL are considered as inadequate to support complex and sophisticated database applications. A new generation of databases has been introduced to satisfy the demands for the support of object-oriented features in DBMSs. The object-relational [40] and object-oriented databases [9] are considered well suited for complex data because of their supports for complex objects, multimedia data and object inheritance etc. [42].

The desire to represent complex objects has led to the development of Object-oriented Database Management Systems (OODBMS). Object-oriented databases employ a data model that supports object-oriented features such as abstract data types (ADTs), polymorphism and inheritance, etc. Through the support of the unique object identifiers (OIDs), objects stored in the databases can be easily identified, and to be shared within a distributed computing network. Object-oriented databases also utilize the power of object-oriented programming languages to provide excellent database programming capability. The combination of OO programming with database technology provides an integrated application development environment. With the use of OO programming languages, such as C++ or Java, object-oriented database applications are able to reduce code size by not having to translate code into a database language such as SQL or using ODBC [5, 35]. As a result, a direct relationship is established between the application data model and the database data model. The strong connection between application and database results in less coding, more natural data structures, and better maintainability and reusability of code.

However, the OODBMS technology is still young and lacks of standards. Until recently, the Object Data Management Group (ODMG) has been founded to propose standards for OODBs. Research is still needed to develop a mathematical object model that supports object-oriented characteristics [38]. Moreover, the incompatibility with the relational databases makes it difficult for most companies to switch to OODBMS because of the heavy investment has been made in RDBMSs.

The development of object-relational database management systems is a hybrid of the RDBMSs and the OODBMSs. ORDBMSs employ a data model that incorporates OO features into RDBMSs, and thus provide a natural and productive way to maintain a consistent structure in the database. All the database information is

stored in tables, but some of the tabular entries may have richer data structures, such as user-defined datatypes and collection types. An ORDBMS supports an extended form of SQL called SQL:1999 (also known as SQL3) that is introduced in the next section. The support from major DBMS vendors and the new OO features make ORDBMSs the market leader. Because of these advantages, object-relational databases are expected to and are making a bigger impact in the market than object-oriented databases [29, 42].

As one of the major RDBMS and ORDBMS vendors, Oracle Corporation introduced Oracle8 in 1999, and released Oracle9i in 2001 that offer a variety of data structures and strong computational powers to create robust database systems. Based on the above observations, we have selected using Object relational database to model the real world objects and their inter-relationships, and employ Oracle9i as the database server for our information system.

3.2 SQL:1999 Standard and Oracle9i

The schema of a database represents the structural and behavioral properties of real-world objects in certain domain. In a traditional relational model, when the structure of the objects and the relationships among them get really complex, the objects must be decomposed into many tuples stored in different tables. To retrieve information from such a database often involves a considerable number of join operations and the performance is hence significantly reduced [10]. However, this problem has been resolved in SQL:1999, the third generation of SQL standards.

In this section, we first look at some new features supported by SQL:1999 and the Oracle9i system, such as new data types and some other OO features, and then we introduce some of the special features provided by the Oracle9i database system.

3.2.1 New OO features supported by SQL:1999 and Oracle9i

SQL:1999 supports two new composite types: ARRAY type and ROW type, which legalize the existence of tables not in first normal form. The ARRAY type allows storing collections of values directly in a column of a database table. The ROW type makes it possible to store structured values in a single column of the database.

In Oracle9i, the VARRAY type and the NESTED TABLE type are supported to represent collections instead of the ARRAY type in SQL:1999. Each element or value for a collection is of the same data type. A VARRAY or a NESTED TABLE type can be used as a column of a table or as an attribute of an object type.

SQL:1999 also supports user-defined data types that are also referred to as structured types. A structured type can be used as a column or as the underlying structured type of a “typed table”. The column definitions of a typed table are derived from the attributes of the underlying type and hence the table and the data type share the same definition and properties of the structure. Each object (tuple) of a typed table is associated with a unique system generated identity that works like an object identifier (OID).

SQL:1999 provides a special type called REF type whose values are OIDs or references of objects stored in typed tables. References are important for modeling relationships and navigating among object instances. A sophisticated usage of the REF types is to follow the reference to access attributes of the referenced object (referee), which is always a row of a typed table. Consequently, the value of a REF type can be considered as a pointer or a link from one tuple to another. This is an important feature that allows us to view the structure of the database as a directed graph whose nodes denote user-defined data types and their attributes, and the references between the types act as edges that indicate the relationships between

objects. The ability to establish one-to-many and many-to-many data relationships without relational foreign keys alleviates the need for relational JOIN operations, because table columns could contain references to rows in other tables. The usage of collection types and references makes it no longer a requirement for the relational database designer to model complex objects in their most atomic components, but allows the real-world objects to have a concrete existence. By dereferencing or following these pointers, one could retrieve rows from other tables without ever using the time-consuming SQL JOIN operator [7].

The pointer notation ($\rightarrow$) is used in SQL:1999 to access the attributes of the OID identified value of the associated structured type. In Oracle9i the dot notation (‘.’) is used instead. The basic usage and the syntax of REF types are demonstrated in the next example. To make this thesis more readable, the following naming conventions are followed in the examples:

- The name of user defined data types all end with ‘_t’ (case insensitive);
- The name of typed tables all have a suffix ‘_tab’ (case insensitive);
- the name of a VARRAY type is named with the suffix ‘_v’ (case insensitive);
- A type of a nested table is named with the suffix ‘_nt’ (case insensitive);

Now consider the relationships among a university, departments of the university, and the professors. To represent this information domain, we have a typed table *professor_tab* of a *professor_t* type with an attribute called *department* which references to a row in a typed table *department_tab* of *department_t* type. The *department_t* also contains a column *university* of REF to an object of *university_t* type in another typed table *university_tab*. Figure 3.1 shows the graphic view over the database schema. For the query *to list the professors’ name and the universities’ name that*

the professor is working for, we can access the attribute *name* of a *university_t* object from a professor object in the *professor_tab* using the SQL:1999 statement:

```
SELECT p.name AND p.department -> university -> name
FROM professor_tab p
```

or the ORSQL statement from Oracle9i:

```
SELECT p.name, p.department.university.name
FROM professor_tab p
```

Note that in the SELECT clause of the two SQL statements, there exists an access path that leads to the attribute *name* of the *university_t* from the professor object, which brings a one-to-one mapping between the schema concepts and the access sequence in the database query. Once this access path has been identified, the generation of the ORSQL statement then becomes a relatively easy job to do.

Another point we want to emphasize is that it is recommended to have bi-directional references defined in the database schema for each pair of related object tuples in order to ensure the accessibility of data. As demonstrated in the above example show in Figure 3.1, every university object keeps a set of references to all its department instances, and each of the department objects keeps a pointer to the corresponding university tuple as well.

SQL:1999 and Oracle9i provide a type model closely aligned with Java's single inheritance. Database designers can specify a given type is either instantiable or not (analogous to abstract types in other programming languages) at the time the type is created. And naturally, at any place (e.g. a column) where a value of some structured type is permitted, a value of any of its subtypes can be used. This provides the substitutability that offered by object-oriented programming technology.

Figure 3.1: Graphical view over the relationships and structure of objects for the University-Department-Professor example

Oracle also provides methods to specialize super types into subtypes when it is applicable. The `TREAT` function of Oracle9i is defined to modify the declared type of an expression to a specified type that is normally a subtype of the declared type. This function attempts to treat a super-type instance as a subtype instance, for example to treat a person as a student. Whether this can be done in a given case depends on whether the person is actually a student (or subtype of student, such as a part-time student). If the person is a student, then the person is returned as a student, with the additional attributes that a student may have. If the person happens not to be a student, `TREAT` returns `NULL`.

For example, assume that we have a database with the *Person_t* as a super type *Person_t* and a subtype *Student_t*. The instances of both types are stored in the table *Person_tab*. The following example shows how `TREAT` is used in a query to retrieve all (and only) *Student_t* instances from object table *Person_tab* of type *Person_t*.

```
SELECT TREAT(VALUE(p) AS Student_t)
FROM Person_tab p;
```

The next example illustrates the use of the `TREAT` function to modify the declared type of a `REF` expression and to access attributes of the referred subtype object, such as the student ID number of a student.

```
SELECT p.name, TREAT(VALUE(p.spouse) AS REF Student_t).sid
FROM Person_tab p;
```

In this example, the *Person_tab* has an attribute *spouse* defined as a reference to a *Person_t* object. The above query retrieves the person's name and the student number of the person's spouse. If the spouse is a student, the value of the student number is returned, otherwise, a `NULL` is returned.

techniques described above, a pure object featured database is built in ORDERS to model the real world objects and their relations. In the underlying database of ORDERS, all the tables are typed tables. The object references, which represent the object relationships, are used anywhere the foreign key constraints may apply.

3.2.2 More about Oracle9i

In addition to the new OO features mentioned above, Oracle9i also provides a comprehensive application development environment. The native support for Java from the DBMS is another important feature that benefits the development of our query reformulation system. Oracle9i has developed its own Java Virtual Machine that is closely integrated with the database mechanisms for high performance and scalability.

The support of JDBC and data dictionary from the Oracle database system is also an important feature. The Oracle static data dictionary views allow application developers to explore the database schema from various levels. For example, an `ALL_xxx` view displays all the information accessible to the current user, including information from the current user's schema as well as information from objects in other schemas, if the current user has access to those objects by way of grants of privileges or roles. Via the use of JDBC and some of the views from the data dictionary, such as `USER_TYPE` and `USER_TYPE_ATTRS`, one is able to retrieve name of the user defined types, their attributes, type of attributes and even the inheritance information.

Chapter 4

Overview of ORDERS

As indicated in Chapter 1, the ORDERS is designed and implemented as a query formulation system for object-relational databases, which interprets the key terms from user queries into database queries. This chapter first describes the system architecture of ORDERS. Then a sample application of ORDERS, the so-called Intelligent Information System, is introduced. At last, several sample queries to the Intelligent Information System are presented to demonstrate the input and output of ORDERS.

4.1 System Architecture

With the supports of the Object-oriented features and the compatibility with the traditional relational databases, ORDBMSs are considered the most appropriate choice for complex database applications [40]. The desire of providing an ORSQL formulation assistant method for object-relational databases has led to the development of ORDERS. The main feature of ORDERS is to semantically reformulate user queries into ORSQL statements. The basic architecture of ORDERS is shown in Figure 4.1

with the query processing components represented in bold rectangles.

Figure 4.1: ORDERS architecture

In this system, the schema graph is defined as a directed graph that is constructed from the database schema to represent the database structure using nodes and edges. It is used in ORDERS as a road map to locate the relevant schema objects, such as the tables, types or attributes that match the user's requests. The system runs over an object-relational database whose schema reflects the ontology of the application domain. After the database is built, the schema graph is created automatically by reading the schema from the DBMS. Then the system is ready to serve as an ORSQL query formulator as listed below:

The ORDERS system takes the three query components that are defined in terms

from the user's vocabulary as: 1) the category, 2) a list of attributes of interests, and 3) the conditions. The first two components are mandatory, where an asterisk (*) can be used as the attribute list to indicate the retrieval of all the available information about the category. The conditions can be optional. The following steps are involved to generate the ORSQL queries:

1. Searching over the table nodes in the schema graph to decide the one that represents the category objects;
2. Using the category table as the start point, get a set of paths that locate all the relevant object types and attributes by traversing the schema graph;
3. Finally generate the ORSQL queries according to the query paths and the ORSQL syntax.

An ontology-based thesaurus is adopted in the schema graph to expand the database structure concepts with related terms or phrases. A fuzzy matching mechanism is used in the process of determining the relevance of tables, types or attributes defined in the database. In the case of more than one set of query paths are identified, a set of ORSQL statements are to be generated as the result. It is left to the user to decide which query meets the request more closely by browsing the well-classified query results. Semantic selection criteria are defined and used in the matching process to prevent the selection of unrelated items. As a result, the more specific the input query is, the more accurate the ORSQL statement will be.

In the implementation of the system, the underlying database is established using Oracle9i that is running on a Sun Enterprise 250 machine under Sun Solaris operating system 2.8. Other components are implemented in JAVA and can be run on any computers that support JDK1.2 (or above) and JDBC with Oracle JAVA library

installed. The system connects to the database via JDBC to read and create the schema graph once it is started.

4.2 A sample application

The ORDERS system has been designed as a back-end interface to an intelligent query system for complex object-relational databases. It can be attached to a front-tier Natural Language (NL) processor that analyzes and decomposes natural language queries into the three query components as the input to ORDERS. There are various Artificial Intelligent (AI) techniques, such as the English pattern recognition and analysis techniques [22], can be used to develop the natural language interface.

Currently we have implemented a sample application of ORDERS called Intelligent Information System. The underlying information source is an object-relational database that covers massive information about countries, cities, states or provinces, and universities in North-America. Relationships between tuples are represented using object references. The Intelligent Information System supports three user query methods, namely the keyword search, the category search, and the semantic search. Figure 4.2 shows the user interface of the Intelligent Information System. The semantic search provides users the ability to issue queries using natural language. A natural language interpreter has been implemented to take queries from end-users, identify the three components of the query, and then pass them to ORDERS to form the corresponding ORSQL statements. After the ORSQL statements are executed, the query results are classified according to the categories and are displayed to the user. English query patterns are recognized by the natural language interpreter so that users are able to write queries not only using the terms they are familiar with but also to present them in a natural fashion. The architecture of Intelligent Information

System is shown in Figure 4.3.

Figure 4.2: User Interface of the Intelligent Information System

Example 1: List the names and websites of universities in Ottawa

Input to ORDERS:

```
category: universities
attributes: names, websites
condition: [city name=Ottawa]
```

ORSQL statement from ORDERS:

```
SELECT x.NAME,x.WEBSITE FROM UNIVERSITY_TAB x
WHERE (UPPER(x.CITY.name ) LIKE UPPER('%Ottawa%'))
```

Example 2: Find professors at Carleton University whose research interests include
Database

Figure 4.3: Architecture of the Intelligent Information System

Input to ORDERS:

```
category: professors
attributes: *
condition: [research interests=Database
            (AND) University name=Carleton University]
```

ORSQL statement from ORDERS:

```
SELECT * FROM FACULTY_MEMBER_TAB x
WHERE (UPPER(x.RESEARCH) LIKE UPPER('%Database%')) and
      (UPPER(x.FACULTY_COLLEGE.UNIVERSITY.name) LIKE UPPER('%carleton%'))
```

Example 3: Find countries whose population is more than 10 million

Input to ORDERS:

```
category: countries
attributes: *
condition: [population > 10000000]
```

ORSQL statement from ORDERS:

```
SELECT * FROM COUNTRY_TAB x
WHERE ( x.PEOPLE.POPULATION > 10000000)
```

Chapter 5

Schema Graph

As shown in the architecture of ORDERS in Figure 4.1 in Chapter 4, the *schema graph* plays an important role in the query processing procedure. Its concrete representation of the domain ontology and the database structure makes it possible to be used as a road map to locate the relevant schema objects, such as tables, user-defined types or attributes. In this chapter we introduce the concept of schema graph and the data structure used in the implementation of ORDERS. The fuzzy matching mechanism that is defined to determine the relevance between the key terms from user queries and the schema objects is discussed as well.

5.1 Schema Graph

In order to take the full advantage of the new OO features and techniques provided by the Object-relational database technology, a pure object featured database is built in ORDERS to model the real world objects and their relations. In the underlying database of ORDERS, all the tables are typed tables. The object references, which

represent the object relationships, are used anywhere the foreign key constraints may apply. The schema graph is designed as a method to concretely represent the database structure and the mappings between the database model and the conceptual model, so that it can be used as the road map to locate relevant schema objects.

In this section we first introduce the concept of the schema graph and its properties. The process of establishing the schema graph is discussed. Some sample database schemas and the corresponding schema graph diagrams are listed to demonstrate this data model.

5.1.1 Concept of the schema graph

As presented in the previous chapters, the object-relational model provides a natural way to represent generic entities or concepts, and their inter-relationships in the information domain in an object-oriented manner. The natural representation of the domain ontology suggests the possibility of mapping the key terms presented in user queries into database concepts from the database schema. In this mapping process, we need a data structure that provides the access to the structure of the database and the ability to dynamically search for the tables and attributes that match the query.

In ORDERS, the structure of the object-relational schema is represented using a directed graph at the database concept level. The *schema graph* is defined as a weakly connected directed graph¹ in which the nodes represent the tables, object types and their attributes (the attributes are shown once for the typed tables and the underlying structured types), and the edges represent the relations between the nodes.

¹A directed graph is weakly connected if the corresponding undirected graph is connected [26]. In an application domain, we assume that the objects are virtually related to each other directly or

From the implementation point of view, the schema graph is a data structure representing a directed graph that contains the structural information (meta-data) about the database schema. In general, two kinds of nodes are represented in the schema graph: the *table nodes* that stand for the tables defined in the database schema; and the *Database Object nodes* that present the structured types and their attributes. The relationships among object types are represented using Database Object nodes and edges. Except the structured types, there are 4 kinds of relationships implied by Database Object nodes:

- Attribute of: indicates the node act as an attribute of the related structured type. In this thesis, we call one of this kind of Database Object nodes as an *attribute node*.
- ISA: denotes the inheritance relation between two structured data types. The subtype inherits all the attributes of the super type, and has its own additional attribute(s).
- IS OF TYPE: a relationship between an attribute node and a structured type node. It indicates that the host type has an attribute that is of the connected structured type.
- Semantic relations: links between structured types to represent information-domain specific semantic relations among the objects.

The last three types of relations, namely the ISA, IS OF TYPE, and semantic relations, act as media between related abstract data types and indicate the relationships exist in the real-world. We call the Database Object nodes between a pair of related structured-type nodes as intermediate nodes.

indirectly. For the rest of the paper, connected means weakly connected.

5.1.2 Example schema and schema graph diagrams

In order to give a graphical view over the schema graph data structure, we now take a look at some examples and the corresponding schema graph diagrams.

Consider a simplified City-University domain that keeps track of city and university objects, the properties of cities and universities, such as their names, the enrollment (number of registered student), and the departments of the university, the professors for each department, etc. There are five user-designed data types defined in this database: *city_t*, *university_t*, *enrollment_t*, *department_t* and *professor_t*; and four typed tables: *city_tab*, *university_tab*, *department_tab*, and *professor_tab*. To keep the example simple, we assume that the president of a university is of the professor type. The schema of the database is shown in Figure 5.1.

Based on the definition of the tables and the user-defined data types, the schema graph of this database is shown in Figure 5.2. The notations used in this diagram are defined as the following: the nodes in bold rectangular shape represent tables, the nodes in plain rectangular shape represent structured types, the ellipses in the diagram act as attributes of the structured types, and the round-cornered rectangles stand for the IS OF TYPE relation or the semantic relationships between structured types that are either of REF types or collection of REFs. Node labels serve as table names, type names or attribute names. For an attribute node, besides the attribute name, the type is also shown in parentheses. The edges in the graph connect the nodes according to the database schema and demonstrate the properties of object types and the relationships between object types. The single arrowhead edges represent the single valued attributes, and the double arrowhead edges indicate multi-valued attributes or collections.

The ISA relationship represents the way in which objects are categorized into

Figure 5.1: Schema of the City-University database

Figure 5.2: Schema graph of the City-University database

subtypes. The objects of subtypes inherit the structure of their generalized superiors (super types) and usually have additional properties not shared by the generic parents. The following is a simple example that illustrates the representation of hierarchical structures using the schema graph. Now consider a university-department-employee domain. In this example, the *employee_t* type has two subtypes, the *staff_t* stands for administration staffs, and the *professor_t* type represents professors or instructors in a university. Figure 5.3 demonstrates the schema for this example.

Figure 5.3: Schema for the University-Employee database

Note in this database, the *professor_t* and the *staff_t* are defined as specialized *Employee_t* by adding extra attributes such as the courses that the professor is teaching or the office location of the staff. There is only one *Employee_tab* defined to store employee objects. The professor and the staff instances are no longer separately stored in a *professor_tab* or in a *staff_tab*, but stored in the *employee_tab* of the super-type

Figure 5.4: Schema graph for the University-Employee database

Employee_t.

Figure 5.4 pictures the schema graph for the University-Employee database. The subtype *professor_t* and the *staff_t* are linked to their super-type *Employee_t* through ISA relationships. The extra attributes of the subtypes are shown in attribute nodes that are connected with the subtypes.

In the schema graph diagrams, the attributes of DBMS system defined types, such as NUMBER, CHAR or VARCHAR2, etc., are referred to as concrete attribute nodes, and are shown in ellipses, such as the *name* node and the *website* node of *university_t* in Figure 5.2. Other attribute nodes and the ISA relation nodes, are the examples of intermediate nodes. This kind of nodes are shown in round cornered rectangles or triangles in the schema graph diagrams, such as the *president* node and *enrollment* node in Figure 5.2, and the ISA nodes in Figure 5.4.

As demonstrated in the above examples, a schema graph is built upon and well conformed to the database schema that represents a common conceptual view over the information domain. In the ORDERS system, the schema graph acts as a road map that can be used to locate the relevant tables and attributes. However, the expected users describe the domain concepts using a richer vocabulary than that is used in the database schema. To semantically identify the tables, types, and attributes that match the user requests is not an easy task. In our system we build a thesaurus that describes the target world in user's terms and is used to map the user queries into concepts from the database schema.

5.2 Thesaurus and Fuzzy Matching Mechanism

Allowing users to define queries in terms of their own vocabulary is an important feature of ORDERS. To support this feature, an ontology dependant thesaurus is established to bridge the gap between the different vocabularies used by the end-users and the database.

Ontology is usually defined as an explicit specification of a conceptualization [13, 23, 24]. The term is borrowed from philosophy, the Greek word *ontos* (being), where ontology is a systematic account of being or existence. When the knowledge of a domain is represented in a declarative formalism, the set of objects that can be represented is called the universe of discourse. This set of objects and the describable relationships among them are reflected in the representational vocabulary. Various thesauri and ontologies, such as WordNet [33] and Sensus [41], are defined to express various relationships between the domain concepts (e.g. synonymy, antonyms etc.), with or without explicit and formal description about the meaning of the concepts. In our research, the ontology is considered to make a general framework for all or most categories encountered by human existence in a specific domain or universe of discourse. Construction of the ontology is usually done by domain experts who have mastery over the specific content of the domain. In our current implementation, a demonstrational ontology is considered based on a general understanding to the example domain.

A thesaurus is commonly referred to as a book of synonyms, often including related and contrasting words and antonyms. In Information Retrieval (IR) systems, a thesaurus is defined as a data structure that consists of (1) a precompiled list of important words in a given domain of knowledge, and (2) for each word in this list, a set of synonyms and related phrases. According to Foskett [6, 20], a thesaurus can be

used (a) to provide a standard vocabulary for indexing or searching; (b) to assist users to locate terms for proper query formulation; and (c) to provide classified hierarchies that allow the broadening or narrowing of the current query request according to the needs of the user. In our model the structure of thesaurus provides a set of terms or concepts that extends the selection of matching database structures.

The definition of a thesaurus can be of a general nature or to be specified to a certain domain of knowledge. The selection of thesaurus should be based on the domain of the underlying database that is designed to reflect the ontology concepts and the purpose of the application. Thus in our system, the thesaurus is constructed as a structure that associates the unique table names, data type names and their attribute names in the schema with a list of synonyms. The list of synonyms of a database concept contains a vocabulary through which the concept that is represented in the database by means of tables, types, and attributes, can be matched with user requests. After the sequence of accessing the relevant schema objects is determined, ORSQL statement is generated to retrieve the values stored in the database. For example, if we build a database whose domain focuses on the universities in North America for non-expert database users, we shall use a domain specific thesaurus to expand the set of terms defined in the schema using related terms obtained from the ontology, for instance, using professor, faculty member, teacher, instructor, and their plural forms as the synonyms for the type node *Professor.t*. When we encounter queries like: *list all faculty members of department of computer science*, the phrase *faculty members* will be recognized as a synonym of professor and to be mapped into *professor.t* in the database. And thus eventually generate the ORSQL statement that retrieves information about all the employees who teach in the department of computer science.

As users do not issue queries in database concepts, we present the *fuzzy matching*

mechanism to map the terms from user requests to schema objects based on the ontology. This method is composed of two parts: one is the use of the ontology based thesaurus we described above to associate the ontology concepts with the corresponding schema concepts, the other is the definition of selection constraints to add constraints to indicate semantic ambiguity and to prevent possible miss-interpretation. The selection constraints are associated with the synonyms from the application domain and are recorded in each schema object node. In the process of determination of the relevance of the schema objects, this information is checked.

Note that in many cases a keyword may be represented in several lists of synonyms and refers to multiple ontology concepts. Sometimes these terms are generally referred to something that are highly related or objects from the same category in the real world, and hence ambiguity is presented. In order to prevent loss of data and miss-interpretation of queries, we add two types of semantic constraints to the synonym list for the schema concepts who have the semantic ambiguity presented. The usage of the thesaurus and the semantic constraints is presented as a *fuzzy matching mechanism* in ORDERS to expand the selection of relevant nodes and to rule out the irrelevant nodes by checking the semantic criteria.

For example, the word *faculty* sometimes refers to a department, and sometimes refers to faculty members. Therefore it can be considered as a synonym both for *department* and for *professor*. Consider the query: list the faculty that offer database courses, the word “faculty” in this query is semantically vague. Without context, it is difficult to determine whether we should interpret the query as to retrieve professors who teach the course or the departments that offer database courses. Our principle for this is to retrieve information as complete as possible, so that both of the above queries should be considered and be reformulated into ORSQL statements. In the

implementation, a “SEE_ALSO” constraint is added to the synonym list of the *professor.t* and of the *department.t* to indicate the ambiguity of the word *faculty*. In this particular case, the string “faculty SEE_ALSO professor.t” is included in the synonym list of *department.t*, and the string “faculty SEE_ALSO department.t” is included in the synonym list of *professor.t*. At the stage of searching for relevant schema concepts, whenever the “SEE_ALSO” constraint encountered, the immediately followed object-type is considered as a relevant node to the query. After the ORSQL statement is generated and executed, the results should be shown in different clusters, as information for professors and information for departments, so that the user can easily navigate through the results by different categories.

The above is a semantic constraint that is defined to broaden the user queries to retrieve information as complete as possible. There are also situations that the ambiguity may lead to semantic errors in the query translation process. For example, consider a query against the schema shown in Figure 5.1 and Figure 5.2: *list the names of instructors in Carleton University*. Take the university as the category, if we search for the node whose synonym list contains the word *instructor*, a synonym of type node *professor.t*, we would end up with a couple of paths that lead to the *Professor.t* as

University.t $\rightarrow$ *President* $\rightarrow$ *Professor.t* $\rightarrow$ *Name*,

and the path

University.t $\rightarrow$ *Departments* $\rightarrow$ *Department.t* $\rightarrow$ *Professors* $\rightarrow$ *Professor.t* $\rightarrow$ *Name*,

between which the second path is the correct one and should be used to generate ORSQL statement (Figure 5.5).

To avoid the problem of inaccurate query path selection, a semantic criteria is added to the synonym list in the node structure. The definition of the semantic

Figure 5.5: Query paths for the query “list the name of instructors in Carleton University”

criteria reflects the semantic meaning of the corresponding user defined structure, and narrows down the selection of matching node during the search. For the above example, the word *president* and its synonyms are defined as the criteria for the selection of the attribute President of the node *University.t*, such that the President attribute node can be selected in the query path if and only if the word *president* or its equivalent terms are particularly specified in the user query.

Through the usage of the fuzzy matching mechanism that is defined in terms of the thesaurus and semantic constraints, the system is capable of handling the queries defined in the terminology that the users are familiar with, so that users are freed from having to write queries in technical jargon, or having to navigate the database schema and specify query paths manually.

5.3 Schema Graph Implementation and Generation of Schema Graph

An adjacency structure [36] is adopted to implement the schema graph in which all the adjacencies are explicitly recorded as pointers in a container for each node. The container can be a vector or an array; the relative order of the elements in the container is unimportant.

Corresponding to the two kinds of nodes represented in the schema graph, two node classes are created to implement the data model. One is the *table node* class that represents a typed table. The other is the *Database Object node* class that represents a structured type or an attribute of a structured type. A table node has the following fields: a NAME field for the name of the table, an ALIAS field for the synonym list, and a TYPE field stores a pointer to the corresponding structured type

node of the table. A Database Object node holds the following information: a NAME field records the name of the type or the attribute, a TYPE field presents the type of the node, such as a user-defined type or a REF type, and a ALIAS field. The alias field in a Database Object node lists the synonyms from the thesaurus for the corresponding type or attribute represented in the schema. The alias of a table node is composed of a collection of the synonyms from the structured-type and the synonyms from all the subtypes along the type hierarchy. The adjacencies between nodes, e.g. the relations between type and their attributes, are represented as pointers to other Database Object node. Thus, a table node is represented by a record:

TABLE_NAME	TABLE_TYPE (a pointer to a Database Object node)	ALIAS
------------	--	-------

And each Database Object node is represented as a record of the form:

Name	Type	Alias	Adjacent_Nodes (pointers)
------	------	-------	---------------------------

In the implementation of ORDERS we associate the related phrases from the thesaurus with the database schema, for example, to predefine the alias information as the COMMENT for each table, structured type, and attribute. At the stage of building the schema graph, the alias field of each node is automatically loaded with other database structural information from the schema.

Two hashtables are used to store the table node objects and the Database Objects for easy access. The names of the nodes serve as the keys. After the set up of the schema graph, which has captured all the necessary schema structural information and the expanded query-related terms, the ORDERS is then ready to serve as an ORSQL query formulator for the underlying object-relational database. The following is the algorithm used to establish the schema graph:

Algorithm schemaGraph():

Input: null.

Output: Schema graph G.

1. Initialize two empty hash-tables: one for table node objects and one for Database Object nodes.
2. Establish a connection to the DBMS via JDBC.
3. Create structured-type nodes by querying the USER_TYPES and the USER_TYPE_ATTRS tables in the data dictionary and retrieve the type names, attribute names, type of attributes. Relationships among structured-type nodes are implemented by adding corresponding intermediate nodes. The value of the *type* field is obtained from the first column of the datatype-mapping table (Table 6.1) according to the type of the attribute defined in the schema. The alias field of each database object node is populated by reading the corresponding COMMENT from the data dictionary.
4. Create table node objects by execute a SQL statement that retrieve all user defined tables from the USER_ALL_TABLES in the data dictionary, and add pointer to the corresponding structured type object. The alias field contains the alias from the structured type and the alias from all its successors.
5. The schema graph is built and the connection to the DBMS is closed.

Figure 5.6 illustrates the structure representation of the schema graph for the *Professor_tab* and its type node defined in Figure 5.1 with values for the fields shown. Note that the SEE_ALSO constraint is presented in the alias field of the *professor_t* node. The adjacent nodes are indicated by curved arrows.

Figure 5.6: Structure representation of the schema graph for the `Professor_tab` and its type defined in Figure 5.1

5.4 Properties of schema graph

The following is a brief summary of the properties presented in the schema graph:

1. The schema graph is a connected directed graph that represents the structure of the underlying database, and is used as a virtual road map for searching relevant objects in the database.
2. The ontology-based thesaurus and semantic selection constraints provide a fuzzy matching mechanism to expand or narrow down the selection of relevant schema concepts.
3. This data model records all necessary schema metadata to generate ORSQL statements.
4. The information represented in the schema graph is either available from the data dictionary or has been pre-associated with the schema objects from the DBMS, so that the set up of the schema graph can be done automatically.

During the query reformulation process, the ORDERS connects to the ORDBMS when it is started. After the schema graph has been established, the connection to the database server is disconnected. Because the schema graph has collected all information that needed to reformulate user queries into ORSQL queries, there is no need to connect to the database server during the query generation process, and thus eliminates the communications with the database management system and reduces the local network traffic between the database server and ORDERS.

Chapter 6

ORSQL Query Reformulation

The use of schema graph has provided a way to match the terms from user queries into relevant schema objects. The next step is to traverse the schema graph to determine the relevant tables, data types, and attributes, as well as the sequences of how to access them. After the accurate and effective access paths have been determined, an ORSQL query generator is invoked to formulate the database queries on the basis of ORSQL syntax. This chapter describes the procedures of the selection of the access paths and the ORSQL query generation process. This chapter is ended with some examples that illustrate the ORSQL reformulation process in different query cases.

6.1 Generation of query trees

6.1.1 Query path and query tree

The generation of ORSQL queries starts with the three parameters passed to OR-DERS, which conform to the three components presented in user queries, namely:

1. the category of the subject of the sentence,
2. the possible attribute list, or an asterisk (*) as to retrieve all the available information about the subject stored in the database,
3. the condition defined in pairs of attribute and value.

Commas (,) are used in the attribute list to separate attributes. The format for the definition of the condition is explained in Section 6.1.3. In the target database query, the value of parameter (1) is mapped to the first table name in the FROM clause, and the possible attributes presented in parameter (2) and of parameter (3) are mapped into the attributes presented in the SELECT clause and the WHERE clause respectively. Take the schema example defined in Figure 5.1 and Figure 5.2 and consider the query: *list the professor names and the name of their departments at Carleton University*. The input to ORDERS are listed below:

- (1) category: professors
- (2) attribute list: name, department name
- (3) condition: [university name=Carleton University]

By searching the schema graph, the category is mapped into the *professor_tab* node, the first *name* attribute is mapped into the *name* attribute node of the *professor_t*, and the *department name* attribute is mapped to the *name* node of the *department_t*, the condition is matched to the *name* attribute of *university_t*, and these nodes are further mapped to the SELECT-FROM-WHERE clauses during the generation of the ORSQL query. The ORSQL query is generated as:

```
SELECT VALUE(x).name, VALUE(x).department.name
FROM professor_tab x
WHERE x.department.university.name LIKE 'Carleton University'
```

In the SELECT clause and in the WHERE clause, there exist paths that indicate the sequence of accessing the attributes of the tables, where the paths start with a table alias x . If we emerge the paths that share a common head, it ends up with a tree that is rooted from the category table as shown below:

As shown in this example, the root is the *professor_tab*, and there are two paths that indicate the sequence of accessing the two attributes in the SELECT clause and one path leads to the attribute in the WHERE clause. This query structure is known as Query by Tree [37]. In ORDERS, a SQL tree is automatically generated by the system through searching the schema graph. The difference between the above tree and the query tree generated by ORDERS is that addition to the sequence of the related schema objects, i.e. the table name and attribute names, the later also records the type of the attributes and other schema structural information that needed in the process of ORSQL query formulation.

A *query path* is as a series of schema graph node objects that indicates the access sequence of the desired destination nodes from the category node. It starts with the category table node, followed by zero or more Database Object nodes and ends with the relevant attribute node as the tail. Given a schema graph G , for simple queries that involve only one structured type, e.g. the type of the category table, a query path P is of the form:

$P = \langle \text{category-table node} \rangle \rightarrow \langle \text{category-type node} \rangle \rightarrow \langle \text{attribute node} \rangle.$

For more complex queries that involve two or more user-defined types, a query path P is in this format:

$P = \langle \text{category-table node} \rangle \rightarrow \langle \text{category-type node} \rangle$
 $\rightarrow \langle \text{intermediate attribute node} \rangle \rightarrow \langle \text{type node} \rangle \rightarrow \langle \text{attribute node} \rangle.$

Where the pattern $\rightarrow \langle \text{intermediate attribute node} \rangle \rightarrow \langle \text{type node} \rangle$ can repeat several times to reach the desired attribute. Note that the query paths always end with an attribute node of a concrete type (DBMS system type) or a structured type. Intermediate nodes only indicate the relationships between type nodes, such as ISA, IS OF TYPE, or relationships presented by object references. The value of intermediate nodes, such as the internal DBMS code of an OID, presents no meaning to end-users and should not be displayed as the result. In the cases that an intermediate node meets the search criteria, the connected object type node is included as the tail of the path.

6.1.2 Construction of Query Trees

In the schema graph, object types are connected via object references and object inheritance. Relationships among objects in the real world are concretely represented in the schema graph through directly connected nodes. Therefore the shortest path between any pair of object types corresponds to the ontology relations among objects in the information domain. The task of obtaining the query tree for the result ORSQL query is thus accomplished by finding the shortest paths that satisfy the fuzzy matching selection constraint described in Chapter 5.

For a schema graph that represents a database with possible cases of inheritance, self-references, and references through circular paths, a schema graph node can be selected more than once in a query path. To ensure the process of searching eventually ends at some point, a system variable *MAXDEPTH* is set to limit the maximum length of a query path that locates the relevant attribute nodes. Since the shortest path between any two nodes indicates the closeness of the relationship between the objects existed in the information domain, when the length of a query path exceeds a certain threshold, the semantic meaning of the relationship represented by the path becomes weak and vague, and the searching of the path becomes meaningless. The *MAXDEPTH* is defined to ensure both the performance and the semantic correctness of the output query paths. The value of the *MAXDEPTH* variable is decided according to the database application and the database structure. For the example schemas presented in this thesis, the integer 9 is set as the value of the *MAXDEPTH*.

The following describes the selection of a query path to an attribute *A* from the user query: Initially the system searches the hashtable that contains the table node objects to find the table that represents the category objects. The *alias* field of each table node is compared to determine whether the corresponding table matches the selection criteria of the category table. After the category table node has been identified, the structured type of the category table is set as the root of the query tree. Then the breadth-first search algorithm is used to select the query path from the schema graph for the attribute *A* in the following way: For each attribute of the root, the *alias* field is checked to decide whether the type or attribute satisfies the selection criteria of *A*. The search ends if an attribute node is found matching the query, and the path that leads to the attribute is retained as part of the tree. Otherwise, the connected type nodes and the attribute nodes are processed until the matching node is found or the length of the query path exceeds the *MAXDEPTH*.

The following is the pseudo code of the *findPath* procedure that is derived from the breadth-first search algorithm. For a given schema graph G , after the category object-type node C has been located, we take the node C as the root node, and use the following algorithm to find the corresponding node of attribute A :

Algorithm findPath(C, A):

Input: Category C , attribute A .

Output: Shortest path to attribute node of A .

Initialize an empty queue q .

Initailize query path $P \leftarrow \langle \text{category-table node} \rangle$.

Add C to the end of P .

Check C against the selection criteria

if C matches the query **then**

return the query path P .

else

$q.\text{enqueue}(P)$.

while q is not empty and length of $P \leq \text{MAXDEPTH}$ **do**

$t = \text{the tail node of } P$.

for each adjacent node n of t **do**

 // Check if n satisfy the selection criteria for attribute A .

if match **then**

 // append node n to path

return $P \leftarrow P.\text{append}(n)$.

else

$q.\text{enqueue}(P.\text{append}(n))$.

$P \leftarrow \text{the front member of } q$.

$q.\text{dequeue}()$.

The output of the algorithm is the shortest path that matches the selection criteria for the attribute A . For each attribute represented in the input parameter (2) or (3), the *findPath* procedure is used to generate the shortest path that leads to the matching node. The query paths are retained to form the query tree, which is used by the ORSQL query formulator to generate the database query. The optimization

of the query trees is achieved naturally during the process of path generation since the less node involved in the query implies the less query operation used in the query process.

The following example demonstrates the algorithm. Consider the query: *name all the professors at Carleton University*, to the City-University database shown in Figure 5.1 and Figure 5.2. The input to ORDERS is:

- (1) category: professors
- (2) attribute list: name
- (3) condition: [university name=Carleton University]

To generate the query paths, the ORDERS first looks up the tables in the table node hash-table and identifies the *professor_tab* as the category table. By following the link, the structured type *professor_t* is located and set as the head of the query paths. Then the *findPath* algorithm is invoked to generate the query paths for each of the attributes from the input. To identify the path to the *name* attribute, the adjacent nodes of the *professor_t* are processed one by one to determine if the selection criteria are satisfied. The alias of the attribute node *name* meets the selection criteria and the path *professor_t* $\rightarrow$ *name* is returned. For the query path to the *university name* in the condition, the *findPath* algorithm is used again to traverse the adjacent nodes of *professor_t*. The *name* node and the *website* node do not meet the selection criteria and are ruled out. By following the links of *professor_t* $\rightarrow$ *department* $\rightarrow$ *department_t* $\rightarrow$ *university* $\rightarrow$ *name* in the schema graph, the node *name* is found match the query. This ends the query tree generation process. Figure 6.1 shows the destination query paths indicated by dashed arrows.

Figure 6.1: Query tree for the query: name all the professors at Carleton University

6.1.3 Handle of Selection Predicates

The query path generation algorithm described in Section 6.1.2 is used to determine the paths that lead to the attributes in the SELECT clause and to the attributes in the WHERE conditions. The result paths for these two clauses however, are represented in different format and are handle in different ways. This is because the query conditions described in the third parameter passed to ORDERS often involve some logical operators and ORSQL comparison operators to indicate the selection predicate that defines the properties the attributes can have.

A selection predicate is defined in the form of $X \text{ op } Y$, where operand X is an attribute name or the equivalent terms, operand Y is an attribute name or the equivalent terms with conditions, or a constant value. The op is an operator from the set $\{<, >, <=, >=, <>, =\}$. The logical relationships between the selection predicates are connected by logical operations, which include the unary operator NOT, and the binary operators AND and OR. The condition of a user query is presented in the format:

$$\langle (\text{NOT}) \rangle X_1 \text{ op } Y_1 (\text{AND}|\text{OR} \langle (\text{NOT}) \rangle) X_2 \text{ op } Y_2 \dots (\text{AND}|\text{OR} \langle (\text{NOT}) \rangle) X_i \text{ op } Y_i,$$

where a pair of the brackets ' $\langle \rangle$ ' enclose one or more optional items, a vertical bar ' $|$ ' represents a choice of two or more options and one of the options should be used in the input of ORDERS.

For predicates whose Y values are attributes, the square brackets ' $[]$ ' are used to surround $X \text{ op } Y$. There could be one or more predicates in the condition. Each predicate is surrounded by a pair of square brackets and is handled individually. The *findPath* algorithm is executed separately to determine the paths that lead to attribute X and to attribute Y . Separate query trees are generated for each X and Y . In the process of ORSQL query formulation, the query trees rooted from the

category object type and from the predicate object type are combined together to form the destination ORSQL statement. The root of the query tree for attribute Y is transformed into a table name in the FROM clause. In typical queries, the two attributes X and Y defined in a selection predicate usually refer to objects of the same type. The type of the two attributes on the two sides of the *op* operator are checked. If the types are not comparable, for example, if attribute X is of a *person.t* type, and Y is found as a *university.t* object, this predicate is considered invalid and is dropped from the WHERE paths. For example, in the query *list the countries whose populations are greater than the population of Canada*, the selection criteria is set to compare the population of countries. The input to ORDERS are expressed as:

- (1) category: countries
- (2) attribute list: *
- (3) condition: [population > population (AND) country name=Canada]

The condition represents the selection predicate. The expression included in the square brackets indicates the conditions applied to the selection of the country objects. In this case, *population* is the operand X, and *population (AND) country name=Canada* is the operand Y. Two separate query trees are generated for X and Y, and are combined in the ORSQL query formulation process. For a complete example, please refer to Example 3 in Section 6.3.

6.1.4 Optimization in query trees

In the query path generation process, the shortest paths that meet the selection criteria for relevant tables or attributes are constructed as the query path, which optimizes the query by involving as least schema objects as possible and pruning out unnecessary query operations. The query tree optimization also comes as a result from the way we choose the value of the input parameters to ORDERS.

The collection types such as VARRAY and NESTED TABLE and other OO features from Oracle provide database designers the ability to mimic the real world relationships in a natural way. However, frequently dereferencing object references stored in VARRAYs or NESTED TABLEs does not yield compelling benefit over traditional table joins and is not recommended by Oracle experts [8]. This problem is avoided in ORDERS by choosing the most decomposed data type in the database structure as the root of the query tree. In a user query, the subject - the start point of query paths from ORDERS - usually tends to be the most nested object with conceptually larger objects specified as the conditions. For example, consider the query *to list the departments of Carleton University*. The subject *departments* in the phrase *the departments of Carleton University* acts as sub-organizations of the Carleton University. In the database shown in Figure 5.1 and Figure 5.2, the *departments* is declared as an attribute of university of a nested table of REFs to department objects. Two ORSQL statements can be issued to generate the same result as indicated in the user query:

```
SELECT *
FROM department_tab x
WHERE x.university.name LIKE 'Carleton University'
```

and

```
SELECT Deref(VALUE(xx))
FROM university_tab x, TABLE(x.departments) xx
WHERE x.university.name LIKE 'Carleton University'
```

The second query suffers overhead by unnesting the nested table and yields extra execution costs over the first one. However, by selecting the *Department_tab* as the

category table, the first ORSQL statement is to be generated as the result query by ORDERS.

The method described above may not generate the most optimal query trees under all circumstances. In the cases that several nested tables from different query trees are involved in a single query, the root of the SQL tree may need to be changed by selecting a query structure that involves as less dereferencing object references in nested tables as possible. We consider the development of a more sophisticated and more comprehensive query tree optimization method as an important part of our future work.

6.2 Generation of ORSQL queries

After the three query components are interpreted into some query trees that embody the relevant schema objects and the access sequences in the SELECT clause and WHERE clause, the remaining work is to form ORSQL statements based on the query paths and the ORSQL syntax. This section introduces the process of ORSQL query generation.

In order to write a valid ORSQL query, we have to know the type of the attributes to determine the operators, the access sequence and the table alias presented in the ORSQL statement in addition to the name of the tables or views, and the attribute names presented in the SELECT attribute list and in the WHERE clause. For example, consider the following ORSQL statement against the university schema shown in Figure 5.1 and Figure 5.2:

```
SELECT VALUE(xx).name
FROM University_tab x, table(x.departments) xx
WHERE x.name = 'Carleton University';
```


In this query we can see that to retrieve the attribute *name* of the departments in the nested table, the operator `TABLE()` is used to flatten the multi-valued attribute *departments* of *University.tab*. This example shows that the attribute type, and the type of the attribute type, which are usually referred to as metadata in a database system, determine the syntax used in the ORSQL query, such as what operation can be applied to each attribute, and the sequence to access the tables and the attributes. Based on this knowledge, a datatype-mapping table is defined to associate the SQL data types with the applicable ORSQL operations. Rules are further defined and are used to formulate ORSQL statements according to the paths obtained from the schema graph.

Table 6.1 shows some of the data type mappings adopted in ORDERS. The first column gives the data type presented in ORDERS. The “SQL:1999 Datatypes” column lists the data types defined in the SQL:1999 standard. The “Oracle9i Datatypes” column lists the SQL types that classified in the Oracle9i database system. The last column lists the SQL operations that applicable to the corresponding type.

In this datatype-mapping table, the values presented in the column of ‘ORDERS datatypes’ are defined based on the classification of Oracle9i built-in datatypes and their applicable SQL operators. Types that share the same query operations are assigned to the same metadata name. For instance, the DBMS types `CHAR` and `VARCHAR2` that stand for fixed or varying length characters and share the same comparison operators in Oracle9i, are presented as ‘`STRING`’ type in ORDERS. During the establishment of the schema graph, the metadata is determined based on the database schema from the Oracle data dictionary and is mapped to the ORDERS datatypes and stored in the *Type* field of a Database Object node record. In the ORSQL statement formulation process, suitable SQL query operations are selected based on the mapping defined in this table to generate query statement. For example,

ORDERS datatypes	SQL:1999 datatypes	Oracle9i datatypes	Description	Oracle9i SQL operations
NUMBER	Numeric	NUMBER, FLOAT, INTEGER	Including integer or non-integer numerics as well as exact or approximate numerics	>, <, =, <>
STRING	Character	CHAR, VARCHAR2	Fixed or varying length characters	=, %, LIKE
DATE	Date	DATE	Date	>, <, =, <>, to_date(<string>, '<format>'), to_char(<date>, '<format>')
REF	Ref types	REF (USER-DEFINED REFERENCE)	REF types to user defined object types	',' (dot notation), VALUE(), Deref()
USER_TYPE	User defined types	USER-DEFINED OBJECT	User defined data types	',' (dot notation), VALUE(), REF()
NESTED_USER, USER_VARRAY	Arrays	NESTED TABLE, VARRAY	Nested table or VARRAY of system (Oracle9i) defined data type	TABLE()
NESTED_REF, REF_VARRAY	Arrays	NESTED TABLE, VARRAY	Nested table or VARRAY of REF to User defined datatype	TABLE()

Table 6.1: ORDERS datatypes, SQL:1999, and Oracle9i datatype mapping

if a `USER_TYPE` or a `REF` type node is encountered, the dot notation ‘.’ is used to access the next attribute node. In case a `NESTED` table or a `VARRAY` node is detected, the `TABLE()` function is used to flatten the collection in the `FROM` clause.

In `ORDERS`, the type inheritance is handled in a different way. In the schema graph, the ‘ISA’ relation is used and recorded in the *Type* field of a subtype node to represent the type inheritance. In such a hierarchical structure, if a subtype node is found relevant to the user query, the path that includes the super type node and the subtype node is recorded as the query path. At the query generation stage, whenever the inheritance relationship is detected along the path, the function `TREAT(VALUE(x) AS [REF] subtype)` is used to cast the super-type instances to subtype objects to access the additional attributes of the subtype.

The following steps outlines the ORSQL query formulation process. After the category table and the query trees for `SELECT` clause and for `WHERE` clause are identified by searching the schema graph, the ORSQL query generator is invoked to formulate the destination queries.

Algorithm queryGenerate(T):

Input: `SELECT` tree S, `WHERE` tree W.

Output: ORSQL query Q.

1. Initialize the `FROM` clause with the category table name that is signed with an x as the table alias.
2. Led by the table alias, the attribute list in the `SELECT` clause is generated based on S. Each path in S is traced starting from the head. The type of every node along the path is checked and the appropriate operator is used according to the datatype mapping table. New table alias is generated by appending a ‘x’ to the current table alias.

3. The access paths presented in the W are handled in the similar way as above, except that the constant value and the comparison operator of each selection predicate is attached to the end of each path. For the predicates in the format $X \text{ op } Y$ that contain the comparison of values from different tuples, the category table of the Y tree is added to the FROM clause, and the letter y is used as the table alias. The Y tree is handle in the similar way as to X tree. The X tree and Y tree are combined together by using the proper comparison operator.
4. Add the key words “SELECT”, “FROM”, and “WHERE” to the access sequence to each corresponding clause, and return the ORSQL query.

In this process, the usage of the punctuations and parenthesis defined in the ORSQL syntax are strictly followed in order to formulate valid ORSQL queries.

6.3 Query Examples

This section illustrates the complete process of query tree selection and the ORSQL query generation through some examples. Four queries are presented in this section among which the first three are issued against the database described in the Figure 5.1 and Figure 5.2. The last example that illustrates the handling of type inheritance based on the schema listed in Figure 5.3 and Figure 5.4.

The first example is a simple query that involves only one table and one user-defined type. The input to ORDERS, the query tree from the schema graph and the result statement are shown to demonstrate the steps involved in the ORSQL generation.

Example 1: List the names and web sites of all universities

Input:

- (1) category: universities
- (2) attribute list: names,web sites
- (3) condition: []

Query tree:

ORSQL query:

```

SELECT x.name,x.website
FROM university_tab x;

```

The following is an example that shows how the semantic fuzzy matching mechanism works in the query tree generation process.

Example 2: List the professors at the School of Computer Science at Carleton University

Input:

- (1) category: professors
- (2) attribute list: *
- (3) condition: [School name=School of Computer Science (AND)
University name=Carleton University]

In this query the word “school” is a synonym for both university and for department. The phrase “the School of Computer Science” is semantically vague by that one can understand it as the name of a stand alone education institute or as a department in a college or university. A “SEE_ALSO” constraint is added to the synonym list to

indicate the vagueness from the ontology and prevent miss-interpretation. In this case, the shortest path from the *professor_t* to the attribute “school” is selected and thus prune out the possibility of visiting the *university_t*. The “SEE_ALSO” constraint of *department_t* starts another search for path to the *university_t* that ends up with two paths which lead to the *university_t* node. The two query trees are shown below, where the second one is considered invalid because of the existence of the duplicated paths and is thus discarded from the query structure.

Query tree:

ORSQL query:

```

SELECT *
FROM professor_tab x
WHERE ((UPPER(x.department.name)
        LIKE UPPER('%School of Computer Science%'))
AND (UPPER(x.department.university.name)
        LIKE UPPER('%Carleton University%')));

```

The next example is a query with complicated selection predicates that involve comparison between values of attributes from different objects.

Example 3: List the universities in Ottawa whose enrollment of fulltime students is more than that of Carleton University

Input:

- (1) category: universities
- (2) attribute list: *
- (3) condition: [city=Ottawa] (AND)
 [enrollment of fulltime students >
 university enrollment of fulltime students
 (AND) university name=Carleton University]

Query tree:

The predicates of this query are composed of two conditions: one is the city of the selected universities is Ottawa, the other is the retrieved universities should have more fulltime students enrolled than that of Carleton University. The second condition involves the comparison between attributes from different university objects and is resulted in two query trees as shown as X tree and Y tree. The Y tree is generated to locate the attributes presented in the Y operand of the second predicate. The roots of the two query trees are signed with the table alias *x* and *y* respectively. Led by the corresponding table aliases, the paths from the query trees are combined together at the stage of reformulating the ORSQL query. The logic operators are preserved in the result statement as they are declared in the user query.

ORSQL query:

```
SELECT *
```

```

FROM university_tab x,university_tab y
WHERE (UPPER((x.city.name) LIKE UPPER('%Ottawa%')) AND
      (x.enrollment.full_time_st > y.enrollment.full_time_st
      AND UPPER(y.name) LIKE UPPER('%Carleton University%')));

```

The following example shows the queries that involve type inheritance. The database structure is listed in Figure 5.3 and Figure 5.4.

Example 4: List the name of the professors and the courses they teach at the Department of Computer Science at Carleton University

Input:

- (1) category: professors
- (2) attribute list: name,courses
- (3) condition: [Department name=Department of Computer Science (AND)
University name=Carleton University]

Query tree:

Note in the database, the type *Employee_t* has two subtypes, namely *staff_t* and *professor_t*. In the schema graph, the alias field of the *Employee_tab* node contains the synonyms from all its subtypes' synonym lists. The ORDERS first compares the category “professors” to the table nodes’ alias, and recognizes the *Employee_tab* node as the category table, then sets the *Employee_t* node as the root of the query tree. The query trees are then constructed as shown above. Finally the ORSQL statement

is generated based on the ORSQL syntax. The TREAT operation is used to cast the *Employee_t* objects into the subtype *Professor_t*. Only the instances that are actually professors are returned in the query result.

ORSQL query:

```
SELECT TREAT(VALUE(x) AS professor_t).name,  
       TREAT(VALUE(x) AS professor_t).courses  
FROM employee_tab x  
WHERE (TREAT(VALUE(x) AS professor_t).department.name  
       LIKE UPPER('%Department of Computer Science%')) AND  
       (TREAT(VALUE(x) AS professor_t).department.university.name  
       LIKE UPPER('%Carleton University%'));
```

Chapter 7

Conclusion

This chapter first summarizes the thesis and identifies the highlights and limitations of this research. Then it concludes with a discussion about the future work.

7.1 Summary

This thesis has introduced an Object Relational Database Query Reformulation System, called ORDERS, as a novel approach to translate application level queries into database queries. The objective of this research is to present a solution to accomplish the semantic mapping of user level concepts into database concepts, find optimal query paths, and automatically generate ORSQL queries for complex object relational database applications, so that users are able to issue queries in the terms from their own vocabularies without worrying about the underlying database structure and the particular query language, and without having to specify all relevant schema objects and the query paths.

The ORDERS system, which is designed as a back-end interface to intelligent

information retrieval systems with complex object-relational data models.

The schema graph, as the most important component in the ORDERS system, is designed and implemented as a directed graph that represents the database structure using nodes and edges. Its concrete representation of the domain ontology and the database structure makes it possible to be used as a road map to locate the relevant schema objects, such as tables, user-defined types or attributes. The usage of data dictionary provided by Oracle9i enables the easy establishment and update of the schema graph, hence provides the flexibility towards easy system revolution collaborated with the revolution of the underlying database. Further more, the schema model records all necessary meta-data that are needed in the ORSQL query generation processes, and consequently reduces the needs to communicate with the database server. As a result, it eliminates the workload of the database system and speeds up the query process.

The ontology-based thesaurus associates the application level concepts with database schema objects, facilitates the concept to concept matching between user requests and the database schema rather than simple keyword to keyword mapping. Based on the ontology-based thesaurus, we have also developed a fuzzy matching mechanism for the semantic concept selection from the database schema. The usage of the *alias* field in the *table node* records and the *database object node* records in the schema graph allows relevant concepts to become associated with database schema objects and to participate in query generation, thus provides the ability of query expansion. The semantic selection criteria impose the constraint encoded in the ontology to prune out possible inaccurate interpretation of queries. The “SEE_ALSO” constraints and the selection criteria take into account the ambiguity represented in the user queries and broaden the selection of relevant database concepts and thus prevent the possible loss of data.

The three query components represented in user queries are passed as input to the ORDERS system. In the query tree generation processes, the breadth-first search algorithm is implemented to traverse the schema graph to locate the relevant schema concepts. The shortest paths to the graph nodes that match the user query and the selection constraints are recognized and are recorded to generate ORSQL queries. The optimization in the query tree is naturally achieved in the implementation of the algorithm and the selection of the category table.

A data-type mapping table is used to associate the Oracle9i system types with the SQL query operations, and is used in the process of ORSQL generation. The query paths generated from the breadth-first search algorithm are combined to form the query trees that construct the frame of the destination ORSQL queries. An ORSQL query generator is implemented to formulate the result queries based on the query trees and the mapping table.

The main features of the ORDERS system are:

1. The system accepts query components in terms from the application domain and interprets them into ORSQL statements using a fuzzy matching mechanism based on an ontology-based thesaurus.
2. A schema graph is established to represent the database schema which simplifies the process of searching for the relevant tables and attributes.
3. Shortest path algorithm is used to automatically find the query paths that construct the frame of the destination ORSQL queries, and to achieve the optimal query generation, so that users are freed from having to specify the paths manually.
4. The ORSQL query reformulation is based on the database schema that is automatically read from the DBMS instead of being preprogrammed. When the

underlying schema changes, the graph is re-established accordingly, so that there is no need to reprogram the whole system.

5. ORSQL queries are automatic formulated based on the query paths and the ORSQL syntax, so that users are freed from having to learn about the ORSQL query language.
6. The sophisticated usage of the new OO features presented in the object-relational technologies, such as object references, collection types, and object inheritance, are handled in the query reformulation system.

We have developed a sample application of ORDERS called the Intelligent Information System. The underlying information source is an object-relational database that covers massive information about countries, cities, states or provinces, and universities in North-America. A user-friendly interface and a natural language interpreter have been implemented to take queries from end-users, identify the three components of the query, and then pass them to ORDERS to form the corresponding ORSQL statements. English query patterns are recognized by the natural language interpreter so that users are able to write queries not only using the terms they are familiar with but also to present them in a natural fashion. The Intelligent Information System can be found at <http://jupiter.scs.carleton.ca>. We have tested the system with more than 300 queries and ORDERS yields satisfactory result.

7.2 Future work

As suggested by experts of Oracle database system, the usage of dereferencing OIDs stored in a NESTED TABLE in an object-relational database does not present compelling benefit over traditional table joins and should be avoided. The query tree

optimization from the current implementation of ORDERS depends on the breadth-first algorithm and the selection of the category table, which intends to start the query from the most nested objects and to eliminate the needs of flattening the nested-tables and the access of the nested table entries. However for complicated queries that involve several query trees this method sometimes tends to be too simple to achieve the best result. More sophisticated methods should be developed to avoid the dereferencing of OIDs in nested tables. The following is a sketchy algorithm for the query tree optimization:

1. Identify the tables that stores the referenced objects.
2. For each of the tables, set the underlying object type as the root of the query tree, and reverse the paths.
3. Compare the number of nested tables involved in the old tree and the new trees.
4. Choose the tree with the minimum number of nested tables as the query plan.

The current implementation of ORDERS supports a limited number of data types provided by Oracle9i. The handling of other data types, such as the BLOB and multimedia data, will be part of the future work.

The future work will also focus on the support of other query operations such as the GROUP BY and ORDER BY operations, and the interpretation of the SQL aggregation operations. The JOIN operation based on the foreign key constraints will be implemented to extend the usage of the schema graph over the traditional relational databases.

The ORDERS has satisfactory result by practice in our research group. More complete evaluation of the ORDERS should also be performed in future work.

Bibliography

- [1] C. Ahlberg and B. Shneiderman. Visual Information Seeking: Tight Coupling of Dynamic Query Filters with Starfield Displays. In *Proc. ACM Human Factors in Comput. Systems '94*, pages 313–317. ACM Press, 1994.
- [2] C. Alhberg, C. Williamson, and B. Shneiderman. Dynamic Queries for Information Exploration: an Implementation and Evaluation. In *Proc. of the CHI'92 Conference*, pages 619–626. ACM Press, 1992.
- [3] J.C. Shafer amd R. Agrawal. Continuous Querying in Database-Centric Web Applications. In *Proceedings of the 9th international World Wide Web conference on Computer networks*, pages 519–531. North-Holland Publishing Co., 2000.
- [4] T.M. Atwood. Object-Oriented Databases: a New Enabling Technology for Electronic Design Applications. *IEEE Electro International*, pages 420 –425, 1991.
- [5] R. Baeza-Yates, editor. *Modern Information Retrieval*. New York, ACM Press, 1999.
- [6] D. Berleson. How oracle 8 will change your life. See http://www.dba-oracle.com/art_chg_ora8.htm, 1997.
- [7] D. Berleson. Oracle9i data model extensions. See http://www.dba-oracle.com/art_9i_data_model.htm , 2002.

- [8] E. Bertino. *Object-Oriented Database Systems: Concepts and Architectures*. Addison-Wesley, 1993.
- [9] E. Bertino and L. Marcos. Object oriented database systems. In O. Daz and M. Piattini, editors, *Advanced Databases: Technology and Design*. Artech House, 2002.
- [10] A.C. Bloesch and T.A. Halpin. ConQuer: a Conceptual Query Language. In *Proc. ER'96: 15th Int. Conf. on conceptual modeling*, pages 121–133. Springer LNCS, 1996.
- [11] A.C. Bloesch and T.A. Halpin. Conceptual Queries Using ConQuer-II. In *Proc. ER'97: 16th Int. Conf. on conceptual modeling*, pages 113–126. Springer LNCS, 1997.
- [12] M. Bunge. *Treatise on Basic Philosophy, Ontology I: the Furniture of the World*. Reidel Publishing Co., 1977.
- [13] K. Doan, C. Plaisant, and B. Shneiderman. Query Previews in Networked Information Systems. A Case Study with NASA Environmental Data. *ACM SIGMOD Record*, 6(1):75–81, 1996.
- [14] A. Eisenberg and J. Melton. SQL:1999, Formerly Known as SQL3. *ACM SIGMOD Record*, 28(1):131–138, 1999.
- [15] R.A. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Addison Wesley Higher Education, 2001.
- [16] D.W. Embley, H.A. Wu, J.S. Pinkston, and B. Czejdo. *OSM-QL: a Calculus-Base Graphical Query Language*. Tech. Report, Dept of Comp. Science, Brigham Young Univ., Utah, 1996.

- [17] C.F. Enguix, J.G. Davis, and A.K. Ghose. Database querying on the world wide web, technical report. See <http://budhi.uow.edu.au/postgrad/carlos/publications/> .
- [18] C.F. Enguix, J.G. Davis, and A.K. Ghose. Database Query on the World Wide Web: UniGuide - an Object-Relational Search Engine for Australian Universities. In *Proc. of the workshop on Reuse of Web Information*, 1998.
- [19] D.J. Foskett. Thesaurus. In K. Sparck Jones and P. Willet, editors, *Readings in Information Retrieval*, pages 111–134. Morgan Kaufmann Publishers, Inc., 1997.
- [20] W. Gietz and C. Dupree. Oracle9i application developer's guide - object-relational features. See <http://www.csis.gvsu.edu/GeneralInfo/Oracle/appdev.920/a96594.pdf> /, 2002.
- [21] J.J. Granacki and A.C. Parker. Phran-span: a natural language interface for system specifications. In *24th ACM/IEEE conference proceedings on Design automation conference*, pages 416–422, Florida, United States, 1987. ACM Press.
- [22] T.R. Gruber. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. In *Proc of International Workshop on Formal Ontology*, Padova, Italy, 1993.
- [23] T.R. Gruber, S. Vemuri, and J. Rice. Model-Based Virtual Document Generation. *Int. Journal of Human-Computer Studies*, 43:907–928, 1995.
- [24] T. Halpin. Conceptual Queries. *Business Rules Journal*, 26(2), 1999.
- [25] A. Hampapur and R. Jain. Video Data Management Systems: Metadata and Architecture. In W. Klas and A. Sheth, editors, *Multimedia Data Management: Using Metadata to Integrate and Apply Digital Media*, pages 245–284. McGraw Hill, 1999.

- [26] F. Harary, editor. *Graph Theory, Addison Wesley Series in Mathematics*. Addison Wesley Publishing Company, 1994.
- [27] A.H.M. Hofstede, H.A. Proper, and P. Weide. Query Formulation as an Information Retrieval Problem. *The Computer Journal*, 39(4):255–274, 1996.
- [28] V. Krishnamurthy, S. Banerjee, and A. Nori. Bringing object-relational technology to mainstream. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Pennsylvania, USA*, pages 513–514. ACM Press, 1999.
- [29] N. Leavit. Whatever Happened to Object-Oriented Databases? *IEEE Computer*, pages 16–19, 2000.
- [30] K. Mahalingam and M.N. Huhns. A Tool for Organizing Web Information. *IEEE Computer*, 30(6):80–83, 1997.
- [31] E. Marcos, B. Vela, J. M. Cavero, and P. Caceres. Aggregation and Composition in Object-Relational database design. In *Proceedings of the Fifth East-European Conference on Advances in Databases and Information Systems*, pages 195–209, 2001.
- [32] R. Meersman. *The RIDL Conceptual Language, Research Report*. Int. Center for Information Analysis Services, Control Data Belgium, 1982.
- [33] G. Miller. WordNet: a Lexical Database for English. *Communications of the ACM*, 38(11), 1995.
- [34] C. Plaisant, B. Shneiderman, K. Doan, and T. Bruns. Interface and Data Architecture for Query Preview in Networked Information Systems. *ACM Trans. on Information Systems*, pages 320–341, 1999.

- [35] B.R. Rao. *Object-Oriented Databases: Technology, Applications, and Products*. McGraw-Hill, 1994.
- [36] E.M. Reingold and W.J. Hansen. *Data Structures*. Little, Brown and Company, 1983.
- [37] N. Rishe, K. Naboulsiand, O. Wolfson, and B. Ehlmann. An efficient Web-based Semantic SQL Query Generator . In *Proceedings. 19th IEEE International Conference on Distributed Computing Systems Workshops on*, pages 23–30. IEEE, 1999.
- [38] C. Schahczenski. Object-Oriented Databases in Our Curricula. *The Journal of Computing in Small Colleges*, 16(1):170–176, 2000.
- [39] B. Shneiderman. Dynamic Queries for Visual Information Seeking. *IEEE Software*, 11(6):70–77, 1994.
- [40] M. Stonebraker and P. Brown. *Object-Relational DBMSs - Tracking the Next Great Wave*. Morgan Kauffman Publishers, Inc., 1999.
- [41] B. Swartout, R. Patil, K. Knight, and T. Ross. Toward Distributed Use of Large-scale Ontologies. In *Proc. of the 10th Workshop on Knowledge Acquisition for Knowledge-based Systems*, 1996.
- [42] M. Wang. Implementation of Object-Relational DBMSs in a Relational Database Course. *ACM SIGCSE Bulletin*, 30(1), 2001.
- [43] T. Wang, P.C-Y Sheu, B. Cummings, and C. Cotman. An Object Relational Database for Brain Aging Research. In *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on*, pages 479 –485, 1998.

- [44] C. Williamson and B. Shneiderman. The Dynamic HomeFinder: Evaluating Dynamic Queries in a Real-estate Information Exploration System. In *Proc. ACM SIGIR '92*, pages 339–346. ACM Press, 1992.
- [45] M.M. Zloof. Query-By-Example: a Database Language. *IBM Syst. J.*, 16(4):324–343, 1977.