NOTE TO USERS

This reproduction is the best copy available.

UMI

Information Network Model Query Processing

By

Xiaopu Song

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

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

January 16, 2010

© Copyright
2010, Xiaopu Song

Library and Archives
Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*
ISBN: 978-0-494-68613-3
Our file *Notre référence*
ISBN: 978-0-494-68613-3

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

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

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

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

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

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

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

Information Networking Model (INM) [31] is a novel database model for real world objects and relationships management. It naturally and directly supports various kinds of static and dynamic relationships between objects. In INM, objects are networked through various natural and complex relationships. INM Query Language (INM-QL) [30] is designed to explore such information network, retrieve information about schema, instance, their attributes, relationships, and context-dependent information, and process query results in the user specified form. INM database management system has been implemented using Berkeley DB, and it supports INM-QL.

This thesis is mainly focused on the implementation of the subsystem that is able to effectively and efficiently process INM-QL. The subsystem provides a lexical and syntactical analyzer of INM-QL, and it is able to choose appropriate evaluation strategies and index mechanism to process queries in INM-QL without the user's intervention. It also uses intermediate result structure to hold intermediate query result and other helping structures to reduce complexity of query processing.

Acknowledgements

I would like to take this chance to sincerely express my gratitude to my supervisor Dr. Mengchi Liu for being such a wonderful advisor. His guidance, encouragement, and immense patience helped me to get through difficult times. I also would like to thank his commitment, direction, insights and financial supports, without which it is not possible for me to finish this thesis.

Especially, I would also like to sincerely thank my parents. They give me their immense support and love, even though they live at the other end of the earth.

My colleagues from the Database Lab supported me in my research work. I would like to thank Jie Hu, Ming Zhong, Dawen Jia, and Qian Chen for sharing their ideas during my research.

Contents

Abstract	iii
Acknowledgements	iv
List of Tables	ix
List of Figures	1
1 Introduction	1
1.1 Objectives	3
1.2 Outline of the Thesis	4
2 Related Works	6
2.1 XML Related Concepts and Query Languages	6
2.1.1 XML	6
2.1.2 XQuery 1.0 and XPath 2.0 Data Model	7
2.1.3 Object Exchange Model	10
2.1.4 Some Query Languages for XML	11
2.2 Evaluation Strategies	15
2.3 XML, Graph Data Models and INM	16

3	Information Network Model	19
3.1	Core concepts	20
3.1.1	Object Classes	20
3.1.2	Relationships	21
3.1.3	Attributes	23
3.2	Induced Role Relationship Class and Context-Dependent Information . . .	24
3.3	Instances	27
3.4	Class and Object Structures	29
3.5	Features of INM	31
4	INM Query Language	34
4.1	Query terms	35
4.1.1	Variables	35
4.1.2	Object Role Terms	35
4.1.3	Element Terms	36
4.1.4	Single Path Terms	37
4.1.5	Multiple Path Terms	39
4.1.6	Class Terms	40
4.1.7	ISA Terms	40
4.1.8	Object Terms	41
4.1.9	Classification Terms	41
4.1.10	Schema Literals	42
4.1.11	Instance Literals	42
4.2	Construction Terms	43
4.2.1	Aggregation Terms	43
4.2.2	General Terms	44
4.2.3	Path Terms	45

4.2.4	Tuple Terms	46
4.2.5	List Terms	47
4.2.6	Grouping Terms	48
4.2.7	Pair Terms	50
4.3	Query	51
4.3.1	Schema Queries	51
4.3.2	Instance Queries	53
4.4	Features of INM-QL	55
5	Implementation	57
5.1	System Overview	58
5.2	Storage	61
5.3	Result Tree	71
5.4	Query Processing	73
5.5	Path Element	76
5.6	Query Optimization	80
5.6.1	Sequential Search Strategy	81
5.6.2	Class-Object Search Strategy	82
5.6.3	Forward Chaining Search Strategy	83
5.6.4	Backward Chaining Search Strategy	86
5.6.5	Hybrid Search Strategy	91
5.6.6	Multiple Literals Search Strategy	97
5.7	Query Output Handler	99
5.8	User Interface	105
5.9	Experimental Results	108
6	Conclusion	113

A BNF of INM-QL	116
Bibliography	119

List of Tables

2.1	Sample XPath Expressions	11
2.2	Sample XPath Expressions with Conditions	12
4.1	Aggregation functions	43
5.1	Size of database as number of objects scales	109
5.2	Query Time and Number of Queried Objects as Number of Objects Scales in Experiment Two	110
5.3	Query Time and Number of Queried Objects as Number of Objects Scales in Experiment Three	111
5.4	Query Time and Number of Queried Objects as Number of Objects Scales in Experiment Four	112
A.1	BNF of Schema Query Expression	116
A.2	BNF of Instance Query Expression	117
A.3	BNF of Result Construction Expression	118

List of Figures

2.1	XML document hospitals.xml	8
2.2	A Node Hierarchy	9
2.3	OEM Graph for XML document	10
3.1	Schema	20
3.2	Hierarchy of Person Class and Its Induced Role Relationship Classes	25
3.3	Person Class and Its Induced Role Relationship Classes	26
3.4	Instance	28
3.5	Hospital Class	30
3.6	Sickroom Class	31
3.7	An Instance of Hospital	32
3.8	Instances of Person	32
5.1	System Architecture	59
5.2	Database Structures	62
5.3	Part of Class file Example	63
5.4	Role relationship and Regular Relationship Structures of Class <i>Hospital</i> . .	64
5.5	Context Dependent Information Structure of Class <i>Internist</i>	64
5.6	Part of Object File Example	65
5.7	Role Relationship Structure of Object <i>OH</i>	66
5.8	Context-Dependent Information Structures of Objects <i>Jack</i> and <i>Jay</i>	67

5.9	Object-Attribute-Relationship File and Its Index	69
5.10	Part of Class-Object File Example	70
5.11	Structures of result tree	72
5.12	Query Processing	74
5.13	Query Manager	75
5.14	A Result Tree Generated for Example 27 (2)	83
5.15	Algorithm of the Forward Chaining Search Strategy for Single Path	84
5.16	Algorithm of the Forward Chaining Search Strategy for Multiple Path . . .	85
5.17	A Result Tree Generated for Example 27 (3)	87
5.18	Algorithm of the Backward Chaining Search Strategy	89
5.19	Algorithm of Backward Chaining Helper	90
5.20	A Result Tree Generated for Example 27 (4)	91
5.21	Algorithm of the Hybrid Search Strategy for Single Path	94
5.22	Algorithm of the Hybrid Search Strategy for Multiple Path	96
5.23	Result trees generated with the hybrid search strategy	97
5.24	Algorithm of the Multiple Literals Search Strategy	98
5.25	Result Trees Generated by the Multiple Literals Search Strategy	100
5.26	A Result Tree of Query 12	104
5.27	Browser console for class file	106
5.28	Browser console for class Hospital	106
5.29	Browser console for object file	107
5.30	Browse console for object OH	107
5.31	INM Query Console	108
5.32	Storage size as number of objects scales	109
5.33	Query Time of Experiment Two	110
5.34	Query Time of Experiment Three	111
5.35	Query Time of Experiment Four	112

Chapter 1

Introduction

Database technology has been developed to effectively and efficiently organize, manage, maintain and retrieve large volumes of data in various memory devices. Research on the underlying structures of databases has led to the development of various data models. The most well-known and widely used data model is the relational data model proposed in [15]. However, the relational data model is not well expressive for many novel database applications. During the past decades, many expressive data models have been developed, such as semantic data models (SDMs), object-oriented data models (OMs), role models (RMs) and graph data models.

The Entity-Relationship (ER) model [14] is considered to be the first semantic data model. It provides a special diagrammatic technique for database design, uses entities to represent real world objects and uses relationships to represent their associations. However, it can only express natural and elementary relationships between objects.

Many other semantic data models [25, 42] and object-oriented data models [8, 2, 46, 10] use high level concepts, such as object identity (oid), generalization, aggregation, classification, instantiation, class hierarchies, and inheritance to represent the real world. They mainly deal with the static aspects of the real world and are not quite suitable for dynamic aspects of objects.

To overcome the limitations of object-oriented data models, various role models have been proposed to capture dynamic aspects of real world objects [44, 47, 4, 20, 45, 18]. They separate object classes and role classes, and enable an object to play several roles. Therefore, the role classes can form hierarchies that support inheritance, as well as object classes. Role classes are also able to express the dynamic classification of objects. The main problem with role models is that they treat roles of objects independently, and all context-dependent information scatters in a hierarchy of objects [31].

Compared to semantic and object-oriented data models, graph data models [27, 26, 21, 6, 41, 24] view the real world as a network of relations and focuses on the interconnection and properties of relations [7]. They apply various basic mathematical definitions of a graph as their formal foundation to capture data interconnectivity. In the graph, an entity represents an object and a relation establishes a connection between two entities. The graph data models use relation to express generalization, compositions, classifications, hierarchy and so on. Since graph data models mainly focus on object interconnectivity and other static aspects of the real world, they are also not well suitable for dynamic aspects of objects.

Real world objects have various natural and complex relationships, through which they are connected [31]. Objects play diverse roles via these relationships. The roles of objects also enable them to possess the corresponding context-dependent properties under their roles. Semantic data models, object-oriented data models, graph data models and role models mainly deal with natural relationships; they either oversimplify or neglect complex relationships. Therefore, they are unable to naturally and directly support various kinds of relationships between objects, between objects and relationships, and between relationships. They are also unsuitable to represent static and dynamic context-dependent information about objects.

Information Networking Model (INM) is a novel data model proposed in [31] to solve the mentioned problem. It naturally and directly supports diverse kinds of relationships

between objects, between relationships and between objects and relationships. It is able to represent various roles that objects play via these complex relationships. It also enables us to represent not only static but also dynamic context-dependent information about objects. In INM, all dynamic context-dependent information is grouped together in objects, rather than being scattered in hierarchies of objects in role models.

In order to effectively express various relationships and context-dependent information at both the schema and instance levels, INM Modeling Language is proposed in [29]. It consists of INM Data Definition Language (INM-DDL) and INM Data Manipulation Language (INM-DML). INM-DDL is designed to naturally model various kinds of relationships between objects, between objects and relationships, and between relationships, and to define schema in the INM database. INM-DML is designed to insert, update, and delete instance in the INM database. INM Query Language (INM-QL) is proposed in [30]. It is specially designed for INM to retrieve information about schema and instance, their attributes, relationships and context-dependent information from the database. INM database management system has been implemented, which uses the thin client/fat server architecture and supports INM-DDL, INM-DML and INM-QL.

This thesis is mainly focused on the implementation of the subsystem that can process INM-QL in INM database management system. INM-QL is a structured query language that always returns accurate query results. The research challenge is how to effectively and efficiently process queries, reduce query complexity, and optimize queries based on the nature of INM-QL.

1.1 Objectives

The overall goal of our work presented in this thesis is to present how we had designed and implemented a subsystem in the INM database management system to effectively and efficiently process INM-QL. The following is a list of detailed objectives of the thesis:

- Build a lexical and syntactical analyzer of INM-QL to produce a stream of tokens, verify that the token stream is syntactically correct, and then construct a valid parse tree for the entire program.
- Introduce an intermediate result structure to hold the intermediate query results and other supporting structures to reduce the complexity of query processing.
- Provide index mechanisms to speed up query processing.
- Design evaluation strategies based on the nature of INM-QL. The system is able to automatically decide and choose appropriate evaluation strategies to process queries without user's intervention.
- Design and implement different algorithms to process query results in the user specified form.
- Conduct experiments to prove that the system is able to answer queries within a reasonable amount of time.

1.2 Outline of the Thesis

The rest of the thesis is organized as follows. Chapter 2 provides some background information for XML and XML query languages, as well as a brief discussion of the related work for our research, which includes query processing and the optimization of XML query languages. Chapter 3 introduces INM. This chapter includes a brief explanation of core concepts with an example of information modeling. Chapter 4 introduces the INM-QL and provides core definitions of instance queries and schema queries and illustrates semantics through rich examples. It also demonstrates how to construct query results with operations, such as order by, aggregate and grouping. Chapter 5 discusses the implementation of the INM database management system. It includes storage, intermediate data structure,

query parser, evaluation strategies and query output handler. This chapter also provides the user interfaces and some experimental examples. Chapter 6 concludes the thesis, lists the contributions of this thesis, and discusses direction for future research.

Chapter 2

Related Works

INM-QL contains path expressions that are fairly similar to path expressions for XML query languages. In order to help understand its path expression, we provide background information about XML and query languages for XML. We also discuss related works on query processing and the optimization of XML query languages.

2.1 XML Related Concepts and Query Languages

2.1.1 XML

Extensible Markup Language (XML) is a markup language for encoding documents electronically [11]. As the amount of information available has increased in recent years, so has the use of XML. An enormous amount of arbitrary data is now stored in XML, both in XML databases and in documents on a file system.

XML mainly consists of many constructs, such as tag, element, attribute and so on. Tag is a markup construct that begins with (<) and ends with (>). There are three kinds of tag in XML: start-tag, end-tag and empty-element tag. The start-tag represents the beginning of every non-empty XML element. For example, <Internist> is a start-tag in

XML. The end-tag represents the end of every XML element that begins with a start-tag. For example, `</Internist>` is a end-tag corresponding to the start-tag `<Internist>`. The empty-element tag is the representation of empty element that has no content. For example, `<empty-content/>` is an empty-element tag. Element is a logic component that begins with a start-tag and ends with a corresponding end-tag. An element can contain other elements as child elements. The content of an element is characters between the start-tag and the end-tag. For example, `<Internist>Jack</Internist>`. Attribute is another markup construct that is composed by a pair of name and value. It can only exist in start-tag and empty-element tag. For example, the element `<Human_Resources office="A-501">Bob</Human_Resources>` has an attribute whose name is office and value is A-501.

Figure 2.1 shows an example of XML document hospitals.xml that contains information about hospitals.

2.1.2 XQuery 1.0 and XPath 2.0 Data Model

The W3C XQuery 1.0 and XPath 2.0 Data Model (XDM) is the data model of XPath 2.0 and XQuery [19]. It defines information about an input to and XQuery processor and all permissible values in the XPath and XQuery languages. It uses nodes to represent XML constructs such as element and attribute. It has six kinds of nodes: element nodes, attribute nodes, document nodes, text nodes, processing instruction nodes and comment nodes. An element node represents an XML element; an attribute node represents an XML attribute; a document node represents an XML document; a text node represents the context of an element; a processing instruction node represents an XML processing instruction; a comment node represents an XML comment.

An XML can be thought of as a hierarchy of nodes. A node can have the hierarchy of nodes that correspond to the XML document hospitals.xml shown in Figure 2.1.

```
<Hospitals>
  <Hospital name="OH" rank="9">
    <Vice_President length="3">
      <Human_Resources office="A-501">Bob</Human_Resources>
      <Medical_Affairs office="A-502">Ben</Medical_Affairs>
    </Vice_President>
    <Doctor>
      <Dentist>Sam</Dentist>
      <Medical_Specialist>
        <Internist>Jack</Internist>
        <Oncologist>Jay</Oncologist>
      </Medical_Specialist>
    </Doctor>
  </Hospital>
</Hospitals>
```

Figure 2.1: XML document hospitals.xml

Figure 2.2: A Node Hierarchy

Figure 2.3: OEM Graph for XML document

2.1.3 Object Exchange Model

The Object Exchange Model (OEM) is a simple and flexible object model proposed in [40]. OEM represents semi-structured data as a directed labeled graph. In this graph, each node represents an object and each outgoing edge of a node represents an element of an object. Each edge is labeled with the name of an element, each node has a unique identifier, each internal node maintains complex values, and each leaf node is labeled with atomic values such as integers, strings, and dates.

The OEM can be used to represent XML. OEM uses Objects and a subobject relationship to represent elements and elements nesting in XML. Figure 2.3 shows the OEM graph for XML document hospitals.xml.

XPath Expression	Description
/hospitals/hospital	Selects all hospital elements of the root element hospitals
/hospitals/hospital/@rank	Selects attribute rank of each hospital
/hospitals/hospital/Doctor	Selects element Doctor of each hospital
//Doctor	Selects all elements named Doctor
/hospitals/hospital/*	Selects all sub-elements of each hospital
/hospitals/hospital/@*	Selects all attributes of each hospital

Table 2.1: Sample XPath Expressions

2.1.4 Some Query Languages for XML

XML is becoming the dominant standard for data representation and exchange on the internet. One prevalent issue in XML research is how to retrieve data from XML documents and to restructure information. Various XML query languages have been proposed, such as XPath [9], XQuery [39] and Lorel [3].

XPath XPath is a query language designed for XML. It is based on the W3C XQuery 1.0 and XPath 2.0 Data Model. Its main purpose is to select elements and attributes from an XML document, while traversing its hierarchy of nodes and filtering out unsatisfied content. For example, consider the XML document hospitals.xml in Figure 2.1 and queries in XPath in Table 2.1.

XPath supports path expression to identify nodes in XML documents, which is a sequence of element type names connected by connectors such as (/) and (//). A path expression describes a linear path to a node that is much like a computer file system path. A forward slash (/) is used to indicate one single step in the path that is relative to the location preceded it. A double forward slash (//) indicates any number of steps and it can be used to navigate all descendants of the context node that preceded it. There are two kinds of

paths: absolute paths and relative paths. A relative path starts with (//) and consists of a sequence of one or more location steps separated by (/); for example, //Vice-President is a relative path. An absolute path starts with (/) and it optionally followed by a relative path; for example, /hospitals is an absolute path. Wildcards (*) is used to select all elements that fulfill the criteria of the preceding path. The (@) is used to specify an attribute in the path expression.

XPath Expression	Description
/hospitals/hospital[@rank="9"]	Selects all "hospital" elements that have a "rank" attribute with a value of "9"
/hospitals/hospital[@*]	Selects all "hospital" elements that have any attribute.
/hospitals/hospital[Doctor]	Selects all "hospital" elements that have a "Doctor" subelement
/hospitals//Doctor[Internist="Jack"]	Selects all "Doctor" elements that have an "Internist" subelement with a value of Jack

Table 2.2: Sample XPath Expressions with Conditions

XPath also has a filter operator "[" to specify selection conditions on the child nodes. It selects all of the nodes that fulfill the criteria and eliminates the set of nodes that cannot satisfy the conditions inside the brackets. For example, consider the XML document hospitals.xml in Figure 2.1 and Table 2.2 gives queries in XPath with conditions enclosed in square brackets.

XQuery XQuery is a query language recommended by W3C for XML documents [22]. It is also based on the W3C XQuery 1.0 and XPath 2.0 Data Model. It enables the user

to select elements and attributes from XML documents, reorganize and restructure them and return the results in structures that the user like. Its primary structure is the FLWOR expression that consists of five parts: for clause, let clause, where clause, order by clause, and return clause. The for clause has an XPath expression and it sets up an iteration through a set of nodes that are returned by the XPath expression. The let clause also has an XPath expression and it provides a convenient way to bind a variable to the entire result of the XPath expression. Therefore, it helps to avoid repeating an expression multiple times. The WHERE clause consists of comparison predicates and logical operators that are used to filter the query results. The order by clause sorts the query result by some certain values. The return clause defines the result format and constructs query results based on the evaluation of the variable bindings.

XQuery supports operations, such as order by, aggregate and group by functions. It provides a large set of functionalities, such as several join types, operators, quantifiers, data types, various predefined functions, conditional expressions, used defined functions and utilization of name space. It also has the ability to perform without predefined schemas.

Query 1. Consider the XML document `hospitals.xml` in Figure 2.1 and the following query in XQuery, which gets the name and rank of all hospitals that are ranked higher than 10 and are sorted by rank of hospital.

```
for $hospital in doc("hospitals.xml")/hospital
let $Vice-Presidents := $hospital/Vice-President
where $hospital/@rank < 10
order by $hospital/@rank
return <hospital>
      {$hospital/@name, $hospital/@rank}
</hospital>
```

In the where clause, XQuery supports quantified expressions “some...in...satisfies...”

and “every...in...satisfies...” to define existential and universal quantifications, respectively.

Query 2. Consider the following query in XQuery, which finds hospitals that have an dentist whose name is “Sam”.

```
for $hospital in doc“hospitals.xml”/hospital
where some $dentist in $hospital//dentist satisfies ($dentist=“Sam”)
return $hospital
```

Query 3. Consider the following query in XQuery, which finds hospitals that have no dentist whose name is “Sam”.

```
for $hospital in doc“hospitals.xml”/hospital
where every $dentist in $hospital//dentist satisfies ($dentist!=“Sam”)
return $hospital
```

LoREL LoREL is a semi-structured data query language in the SQL/OQL style [34]. Its underlying data model is OEM. It consists of three clauses: select clause, from clause and where clause. It supports a declarative path expression to traverse graph data and retrieve meaningful results. For example, *hospitals.hospital x* is a simple path expression, where *hospitals* is the name of an object and *hospital* is a label. This path expression denotes that variable *x* ranges over all *hospital* labeled subobjects of the object *hospitals*. Variable *x* ranges over an empty set if *hospitals* is a leaf node or if *hospital* is not an outgoing label of *hospitals*.

Query 4. Consider the graph in the Figure 2.3, and following query in LoREL, which finds all *hospital* labeled subobjects of the object *hospitals* that have an internist whose name is “Sam”.

```
select H
from hospitals.hospital H
where exists x in H.Doctor.Internist: x=“Sam”
```

2.2 Evaluation Strategies

In order to effectively and efficiently process INM-QL, we have studied several approaches related to SQL query optimization and XML query processing and optimization [13, 23, 1, 39, 12, 33, 37]. There are three main strategies regarding XML query optimization in Lore system: top-down strategy, bottom-up strategy and hybrid strategy [34].

The top-down strategy starts at an object that is the top of the OEM graph, follows edges whose names are in the path expression in a forward manner and matches predicates in the condition. It leads to a depth first traversal of the graph.

For example, the top-down strategy is the most straightforward approach to process the Query 4. It starts at an object *hospitals* that is the top of the OEM graph and fully explores all *hospital* subobjects of *hospitals*. It then follows the path expression and explores all *Doctor* subobjects of *hospital*. For each subobject *Doctor*, it searches for a subobject *Internist* of *Doctor*, whose value is “Sam”.

The bottom-up strategy traverses backward the path expressions. It starts with objects that are at the bottom of the OEM graph and navigates from child to parent in the graph to match path expressions in reverse. The bottom-up strategy provides another way to process Query 4. It first searches through all objects for objects that satisfy the predicate “ $x = \text{Sam}$ ”. If there is one object that satisfies the predicate, it traverses backward from the found object to its patient objects, so as to match the path expression from the end to the beginning. So the reverse path expression that it matches is “Internist.Docter.hospital.hospitals”. The advantage of bottom-up strategy is that it always starts with objects that satisfy the conditions and avoids needlessly exploring unsatisfying objects. However, the bottom-up strategy performs fairly poorly when there are many objects that satisfy predicates in the condition but very few of those objects match the path expression.

The hybrid strategy separates a path expression into two segments and simultaneously processes the former segment with the top-down strategy and the latter segment with the

bottom-up strategy. It creates a temporary result set of objects that satisfy the former segment of the path expression and also creates another result set of objects that satisfy the latter segment. It then joins two result sets of objects to satisfy the complete path expression. If the fan-out degree of matching path expression with top-down strategy and the fan-in degree of matching the path expression with bottom-up strategy are both very large, then the hybrid strategy is optimal. Thus, each strategy may not be applicable to or perform fairly poorly on some cases, but can be particularly efficient for other cases.

2.3 XML, Graph Data Models and INM

Generally, XML is represented using a tree-like structure. However, it supports a referencing mechanism among elements to allow cycles among data nodes. As a result, it is able to support a graph-like structure and simulate semi-structured data. In XML, there is no separation between data and schema. The hierarchical structure of data is described by the data itself. In addition, XML represents data as different nodes and does not support the concept of object.

Compared to XML, in the graph data models [27, 26, 21, 6, 41, 24], an entity represents an object and a relation establishes a connection between two entities. Graph data models view data in the real world as a network of relations with properties. They separate the data and the schema, and describe the hierarchical structure of the data in the schema graph. They use relation to explicitly express generalization, compositions, classifications and hierarchy.

INM views the real world as a network of objects that are connected via natural and complex relationships. Compared to XML and graph data models, INM supports graph-like structures with hierarchies of complex relationships among objects. Objects are able to play various roles through these complex relationships. The roles of objects also enable them to possess the corresponding context-dependent properties under their roles. INM

represents not only static but also dynamic context-dependent information about objects. Moreover, all dynamic context-dependent information is grouped together in objects, rather than being scattered in hierarchies of objects in role models.

XPath, XQuery and Lorel are used to explore the tree-like structure and to retrieve information from XML documents. XPath and XQuery support path expression that is a sequence of element type names connected by connectors, such as (/) and (//). They provide (@) to distinguish attributes. Additionally, XQuery has a return clause to process query results, supports operations, such as order by, aggregate, and group, and provides a large set of functionalities, such as quantifiers and data types. Lorel is much like SQL/OQL and supports a path expression that is a sequence of labeled edge names connected by connector (.).

WebSQL is a query language for World Wide Web in SQL style [35]. It is based on a virtual graph model that views the Web as a network of documents and a set of relations among them. The web is viewed as a finite database in this model. However, WebSQL can still have terminating behaviors, because of the dynamic nature of the Web and the lack of concurrency control [36]. It uses multiple index servers to accelerate query processing and the user does not need to know about multiple index servers. In other words, they are transparent to the user. It also uses the idea of query locality to evaluate query cost and reduce the complexity of query processing.

Compared to XPath and XQuery, INM-QL uses the same connectors like (/) and (//) in the path expression, but is designed to explore the graph of objects connected with various relationships and to retrieve information about the schema and the instance, their attributes, relationships and context-dependent information. It has several evaluation strategies that are based on its nature, some of which perform very similarly to evaluation strategies of Lorel. It takes advantage of the index mechanism to accelerate query processing. The evaluation strategies, index mechanism, and other helping structures are designed to reduce complexity of query processing and they are transparent to the user. Compared to WebSQL,

INM-QL can always terminate within a reasonable amount of time because of the nature of INM and INM-QL. Moreover, it provides a result construction expression to process query results in the user specified form and supports operations such as order by, aggregate and group.

Chapter 3

Information Network Model

This chapter introduces Information Networking Model (INM), which includes core concepts, induced role relationship classes and context-dependent information, instances, and features. It shows how INM naturally and directly supports various relationships and allows context-dependent representation.

Let us take hospital information modeling as an example. Suppose a hospital involves several kinds of people, including a vice-president, a doctor and a patient. A vice-president can be specialized into vice-president human resources and vice-president medical affairs, and can have a start year, office and length as its attributes; inversely, a person can be assigned vice-president position at a hospital, which means that he or she plays a role of vice-president. A doctor can have a number of specializations, such as a dentist and a medical specialist. A medical specialist can also have a number of specializations, such as an internist and an oncologist. A doctor may take care of several patients and manage some sickrooms; inversely, a person may work in a hospital as a doctor with the status of dentist or internist. A patient uses a sickroom and is taken care of by a doctor; inversely, a person can play a role as a patient who has the health status of a patient in a hospital. A sickroom has a room number and a number of specializations such as an intensive care unit; it is managed by doctors and is used by patients. Figure 3.1 illustrates the schema of

Figure 3.1: Schema

hospital information modeling by using the INM; we introduce the schema and discuss the details in Section 3.1.

3.1 Core concepts

Now we use the above example to introduce the core concepts of INM.

3.1.1 Object Classes

An object class is a set of objects that share common attributes and relationships, which are used to describe static aspects of the real world entities. They can have static subclasses

that form hierarchies of object classes, and inherit attributes and relationships from their super classes, as in object-oriented data models and role models.

Figure 3.1 shows that we treat *Hospital*, *Person*, *Sickroom*, and *Intensive Care Unit* as object classes, which are denoted graphically with rectangles. *Intensive Care Unit* is a static subclass of object class *Sickroom*. Therefore, *Sickroom* and *Intensive Care Unit* form a object class hierarchy.

3.1.2 Relationships

A relationship is an association among objects [14]. INM supports four kinds of relationships: regular relationships, role relationships, context relationships and context-dependent relationships.

A regular relationship is used to represent how two objects are related with each other, and it functions similarly to the relationship in the ER model [14], semantic model [42] and object-oriented models [5, 46]. It can have a inverse relationship that is also a regular relationship. For example, *Hospital* has a regular relationship *offers* with *Sickroom*; inversely, *Sickroom* has a regular relationship *offerBy* with *Hospital* in Figure 3.1.

A role relationship is not only used to represent the relationship from an object *A* to another object *B*, but is also used to express the role that object *B* plays through the role relationship in object *A*. Objects *A* and *B* are known as the source object and the target object of the role relationship, respectively.

For example, we use ellipses to graphically denote role relationship in Figure 3.1, such as *VicePresident*, *VicePresident-HumanResource*, *VicePresidentMedicalAffairs*, *Patient*, *Doctor*, *Dentist*, *Medical Specialist*, *Internist*, and *Oncologist*. Those mentioned role relationships are role relationships from source class *Hospital* to target class *Person*. They not only are relationships that connect objects in *Hospital* to objects in *Person*, but are also the roles that objects in *Person* play in objects in *Hospital*.

A role relationship is allowed to possess attributes and context-dependent relationships. As object classes, it and its role sub-relationships form a role relationship hierarchy. They override or inherit normal attributes from their role super-relationships both at the schema level and at the instance level. A role relationship induces a role relationship class and its context-dependent attribute and relationships are used to form context-dependent information. For example, $VicePresident \rightarrow \{VicePresident-HumanResource, VicePresident-MedicalAffairs\}$ is a role relationship hierarchy in Figure 3.1. *VicePresident* has three attributes: *length*, *office*, and *length*. The first two attributes are normal attributes, whereas the third attribute is a context-based attribute. *VicePresident-HumanResource* and *VicePresident-MedicalAffairs* inherit attributes *length* and *office*. Third attribute is used to generate context-dependent information of induced role relationship classes that correspond to role relationships, see Section 3.2.

A context relationship is used to represent the inverse relationship and identification of a role relationship. The context-dependent information of a role relationship class is automatically generated in terms of the context relationship and the identification of a role relationship. A context-dependent relationship is used to represent the relationship from an object in a certain context to another object that can be in another context or nothing. A context-dependent relationship is used as a nested part in the context to form context-dependent information.

For example, in Figure 3.1, the role relationship *Doctor* has a context-dependent relationship *manages* with object class *Sickroom*, which indicates that if a person becomes a doctor in a hospital, he or she may have a relationship *manages* with sickrooms. The role relationship *Internist* has a context-dependent relationship *takesCare* with the role relationship *Patient*; inversely, *Patient* has a context-dependent relationship *takenCareBy* with *Internist*, which indicates that if a person becomes an internist in a hospital, he or she may have a relationship *takesCase* with a patient; inversely, if a person becomes a patient in the same hospital, he or she may have a relationship *takenCareBy* with an internist. *takesCare*

and *takenCareBy* are defined on the role relationships *Doctor* and *Patient*, and are used to illustrate the associations between different persons in different contexts. When a person changes his or her role, the context-dependent relationship should be deleted from context-dependent information and his or her new context-dependent information should be automatically generated according to his or her new role.

3.1.3 Attributes

Both objects and role relationships are allowed to have attributes. Because of role relationships, not all of their attributes should be dealt with in the same way. Some attributes are just as normal and usual attributes that objects have, whereas other attributes are dependent on roles that objects play. INM supports two kinds of attributes: normal and context-dependent.

A normal attribute is used to describe a common property of either an object or a role relationship. A normal attribute is only dependent on either an object or a role relationship under an object. For example, in Figure 3.1, *name*, *age*, and *gender* of object class *Person* are normal attributes, and their values are dependent on objects of *Person*. Attributes *office* and *length* of role relationship *VicePresident* under *Hospital* are also normal attributes that are used to describe properties of role relationship *VicePresident*. Values of *office* and *length* are only dependent on instances of *Hospital*, and they may remain the same no matter who is vice-president, vice-president of human resources, and vice-president of medical affairs.

A context-dependent attribute is used to represent a dynamic property of an object that plays a certain role. It is used as a nested part to form context-dependent information in the context. For example, *startYear* is a context-dependent attribute of the role relationship *VicePresident* that has the identification position in Figure 3.1. The values of *startYear* are dependent on individuals who are appointed to the position.

3.2 Induced Role Relationship Class and Context-Dependent Information

A role relationship class and its context-dependent information is induced from a role relationship. It has the same name as the role relationship. It is a subclass of the target class of the role relationship. Objects that belong to the role relationship class play roles through the corresponding role relationship in the context of the source class.

Role relationship classes can also form a hierarchy that is very similar to the hierarchy of their corresponding role relationships. Moreover, their target class is the super class of the root of the role relationship class hierarchy that supports inheritance like hierarchy of object classes. The root just inherits or overrides normal attributes, regular relationships, and role relationships from the target class if the target class is an object class. Its complete context is composed of context relationship and identification of its corresponding role relationship. Context-dependent attributes and relationships specified on the role relationship are nested in the context of the role relationship class and form its context-dependent information. The root inherits or overrides not only normal attributes, regular relationships, role relationships but also context-dependent information from the target class, if the target class is a role relationship class. The final context-dependent information of the root is composed of the context of the target class and current context-dependent information of the root. If a role relationship class is not the root in a role relationship class hierarchy, it inherits or overrides all attributes, relationships, and context from its super class. Its final context-dependent information is composed of its context and context-dependent attributes and relationships of its corresponding role relationship.

Figure 3.2 demonstrates hierarchy of object class *Person* and induced role relationship classes. For example, role relationship class *Doctor* is a subclass of object class *Person*; meanwhile, object class *Person* is the target class of role relationship *Doctor*. Role relationship class *Doctor* has two role relationship subclasses: *Medical Specialist* and *Dentist*.

Figure 3.2: Hierarchy of Person Class and Its Induced Role Relationship Classes

The role relationship class *Medical Specialist* also has two role relationship subclasses: *Internist* and *Oncologist*.

Figure 3.3 illustrates final class structures of object class *Person* and induced role relationship classes, which include attributes, relationships, and context-dependent information. We discuss the details about class structure in Section 3.4. For example, role relationship *VicePresident* has an identification *position* and its target is an object class; thus, the context of role relationship class *VicePresident* is composed of identification *position*. Context-dependent attribute *startYear* is nested in the context of role relationship class *VicePresident* to form the context-dependent information. Role relationship *Doctor* has a context relationship *worksIn* and its target is an object class; thus, the context of role relationship class *Doctor* is composed of context relationship *worksIn*. Context-dependent attribute *D#* and two context-dependent relationships *takeCare* and *manages* specified on role relationship *Doctor* are nested in the context of corresponding role relationship class *Doctor* to form the context-dependent information. Role relationship *MedicalSpecialist* has a context relationship *worksIn* and its target is an object class; thus, the context of role relationship class *MedicalSpecialist* is composed of context relationship *worksIn*. Role relationship class *MedicalSpecialist* inherits a context-dependent attribute *D#* and

Figure 3.3: Person Class and Its Induced Role Relationship Classes

two context-dependent relationships *takeCare* and *manages* from role relationship class *Doctor*, and overrides context-dependent relationship *manages*. Those context-dependent attributes and context-dependent relationships are nested in the context of corresponding role relationship class *MedicalSpecialist* to form the context-dependent information. Role relationship *Internist* has context relationship *worksIn* and identification *status*; thus, the context of role relationship class *Internist* is composed of context relationship *worksIn* and identification *status*. Role relationship class *Internist* inherits a context-dependent attribute *D#* and two context-dependent relationships *takeCare* and *manages* from role relationship class *MedicalSpecialist*, which are nested in its context to form the context-dependent information.

3.3 Instances

Based on the schema shown in Figure 3.1 and core concepts introduced above, we demonstrate eight networked objects in the instance shown in Figure 3.4. *OH* is an object of object class *Hospital*, and *ICU-01* is an object of object class *Intensive Care Unit*. For role relationship classes, *VicePresident-HumanResources* has an object *Bob*, *VicePresident-MedicalAffairs* has an object *Ben*, *Patient* has two objects *Ann* and *Ben*, *Internist* has an object *Jack*, and *Oncologist* has an object *Jay*. All objects are related with various relationships; complex context-dependent information of different objects is dependent on different contexts.

In INM, all information including the complex context-dependent information of a real world object is stored together in one instance, rather than being scattered in a hierarchy of objects, as in role models. In Figure 3.4, *OH* has a role relationship hierarchy *VicePresident* $\rightarrow$ {*VicePresident-Human Resources*, *VicePresident-Medical Affairs*} with *Bob* and *Ben*. For *Bob*, he has a position *VicePresident-Human Resources* in *OH*, and in this context he has an attribute *startYear* with value 2007. Hence, the following expresses the complex

Figure 3.4: Instance

context-dependent information of *Bob*:

position:OH.VicePresident-HumanResources[startYear:2007].

For *Ben*, he has a position *VicePresident-MedicalAffairs* in *OH*, and in this context he has an attribute *startYear* with value 2008. He is also a patient in *OH*, and in this context has a patient number with value 002. Hence, the following expresses the complex context-dependent information of *Ben*:

position:OH.VicePresident-MedicalAffairs[startYear:2007], health:OH.Patient[P#:002].

Moreover, *OH* has a role relationship hierarchy $Doctor \rightarrow \{Medical\ Specialist \rightarrow \{Internist, Oncologist\}\}$ with *Jack* and *Jay*; inversely, both *Jack* and *Jay* have context relationships *worksIn* with *OH*. For *Jack*, he works in *OH* with status *Internist*, and in this context he has attribute *D#* with value 001, takes care of *Ann*, and manages *ICU-01*. Hence, the following expresses the complex context-dependent information of *Jack*:

worksIn:OH[status:Internist[manages:ICU-01, takeCare:Ann, D#:001]].

For *Jay*, he works in *OH* with status *Oncologist*, in this context he has attribute *D#* with value *002*. Hence, the following expresses the complex context-dependent information of *Jay*:

worksIn:OH [status:Oncologist[D#:002]].

3.4 Class and Object Structures

From an object-oriented perspective, we use class and object structures to represent the schema and the instance in Figure 3.1 and Figure 3.4, respectively. They are able to retain all information in the schema and instance without losing any information.

The class and instance structures are tree structures that consist of following five kinds of nodes:

- Class-object nodes
- Role relationship nodes
- Target nodes
- Context relationship target nodes
- Identification target nodes

A class-object node represents a class or an object and contains its attributes and regular relationships. A role relationship node represents a role relationship in a class or an object and contains normal attributes. A target node represents a target of a regular relationship or a role relationship. It is a reference to another class or object and maintains target name and its ID. A context relationship target node represents a target of a context relationship. It is a reference to another class or another object. It not only stores target name and ID but also stores identification or nested context-dependent properties. An identification target node

Figure 3.5: Hospital Class

represents a role that class or object plays. It is a reference to a role relationship in another class or another object. Therefore, it not only stores name and ID of a role relationship, but also stores the context relationships or nested context-dependent properties.

We can use diagrams to demonstrate structures of classes and objects. Be aware that we use rectangle, ellipse, parallelogram, circle and round rectangle to graphically denote class-object node, role relationship node, target node, context relationship target node, identification target node in following graphical examples.

Figure 3.5 demonstrates the structure of object class *Hospital*, which includes attributes, various relationships, and targets of relationships. Hierarchies of role relationships in the object class become role relationship nodes, and target nodes contain class name and the automatically generated ID of object class *Person*.

Figure 3.6 demonstrates the structures of object class *Sickroom*. It has three inverse relationships: *manageBy*, *usedBy*, and *offerBy*, with object classes *Doctor*, *Patient* and *Hospital*, respectively. It also has a subclass *Intensive Care Unit*.

Figure 3.7 demonstrates the structure of object *OH* that belongs to object class *Hospital*, which includes attributes with values, various relationships, and targets of relationships.

Figure 3.6: Sickroom Class

Hierarchies of role relationships in *OH* become role relationship nodes. Target nodes of role relationship *VicePresident-HumanResources* and *VicePresident-MedicalAffairs* store instance names and automatically generated IDs of objects *Bob* and *Ben* respectively.

Figure 3.8 demonstrates the structures for *Bob*, *Ben*, *Ann*, *Jack*, and *Jay* of object class *Person*, including attributes, various relationships, and context-dependent information. For example, *Jack* has context relationship *worksIn* and we use a context relationship target node to refer its target *OH*. We also use an identification target node to refer to the target of identification *status*, and store a nested attribute *D#* with value *001* and two nested context-dependent relationships *takeCare* and *manages*. Two target nodes refer *Ann* and *ICU-01* as targets of *takeCare* and *manages* respectively.

3.5 Features of INM

We summarize the features of the INM as follows:

- One of the novel features of INM is the introduction of role relationship, context

relationship, and context-dependent relationship. These three relationships are used to naturally and directly model dynamic and many-faceted aspects of real world objects. Role relationships are able to form hierarchies that support inheritance like object classes.

- INM introduces a new way to treat two kinds of attributes: normal attributes and context-dependent attributes. A context-dependent attribute is used to describe a dynamic property of an object that plays a certain role.
- Role relationships induce corresponding role relationship classes and their context-dependent information at the schema level. Role relationship classes are able to form a hierarchy that also supports inheritance with overriding.
- Context-dependent information is automatically generated in terms of context relationship and identification at the instance level. The complete context consists of context relationship and identification of role relationship. Context-dependent attributes and context-dependent relationships are nested in the context to form final context-dependent information. All of the information including complex context-dependent information of a real world object is stored together in one instance, rather than being scattered in a hierarchy of objects, as in role models.

Chapter 4

INM Query Language

This chapter introduces in detail the design and functionality of INM-QL that can be classified into two kinds: *schema queries* and *instance queries*.

Schema Queries They are used to retrieve the information about classes and their subclasses, superclasses, attributes, relationships and context-dependent information. Classes are linked using class identifiers via all kinds of relationships. From a class, we can easily get its semantic related classes via a variety of relationships.

Instance Queries They are used to retrieve the information about objects, their attributes and relationships. Instances are linked using object identifiers via all kinds of relationships. From an instance, we can easily get its semantic related instances via a variety of relationships.

In INM-QL, there are two kinds of terms: query terms and construction terms. Query terms and construction terms can be used in both schema queries and instance queries. The examples are based on schema in Figure 3.1 and instance in Figure 3.4.

4.1 Query terms

Query terms are used to form query expressions, to match classes, objects, and their components, and to bind variables to values. A query term is ground if it has no variables, is general if it just has variables and is mixed if it has one variable at least.

4.1.1 Variables

Variables in query expression are used to bind classes, objects and various parts of classes and objects based on their locations to get information. A variable starts with a dollar sign (\$) followed by an alphanumeric string. One primary feature of INM-QL is that variables are all logical variables that are place holders and have no types. They are additional all single-valued, and are flexible and easy to use in practice.

4.1.2 Object Role Terms

An *object role term* is used to match an object role property in context-dependent information. It is an expression of the following form:

$$O.R \quad (4.1)$$

where O is either a variable, a class name, or an object name, and R is either a variable or a relationship name.

Example 1. The following object role terms can be used in schema queries:

Hospital.VicePresident	Hospital.\$X
\$X.VicePresident	\$X.\$Y

The following object role terms can be used in instance queries:

OH.VicePresident-HumanResources	OH.\$X
\$X.VicePresident-HumanResources	\$X.\$Y

4.1.3 Element Terms

An *element term* is an expression of the following form:

$$N : * T \quad (4.2)$$

where N is either a variable, an attribute name, a relationship name or an identification name, and T is either a variable, a type, a class name, a value, an object name, an object role term, a set of values, a set of object names or a set of ground or mixed object role terms. N or T can be omitted, but not at the same time. If T is either a variable, an object name, or a set of object names, wildcard (*) can appear; otherwise, it must be omitted.

In schema queries, an element term is used to match either an attribute definition, a relationship definition, or part of context-dependent information definition. In instance queries, it is used to match either an attribute expression, a relationship expression, or part of context-dependent information expression. N matches either an attribute name, a relationship name or an identification name. Correspondingly T in schema queries matches a type, a class name or a role property and T in instance queries matches a value, an object name, an object role property, or just a role property. Due to the hierarchy of role relationship, wildcard (*) denotes that T matches not only target names of role relationship N , but also target names of role sub-relationships of N .

Example 2. The following element terms can be used in schema queries:

age:Int	\$X:Int	age:\$X	\$X:\$Y
age	:Int	VicePresident	:Person
VicePresident:Person		VicePresident:\$X	\$Y:Person
position:Hospital.VicePresident		position:Hospital.\$X	position
position:\$X.VicePresident		position:\$X.\$Y	:Hospital.\$X
\$X:Hospital.VicePresident		\$X:\$Y.\$Z	:\$Y.\$Z

The following element terms can be used in instance queries:

age:45	\$X:45	age:\$X	\$X:\$Y
age	:45	\$X:*\$Y	:Bob
position:OH.VicePresident		position:OH.\$X	position
position:\$X.VicePresident		position:\$X.\$Y	:OH.\$X
\$X:OH.VicePresident		\$X:\$Y.\$Z	:\$Y.\$Z
VicePresident-HumanResources		\$X:Ben	:{Ben,Bob}
VicePresident-HumanResources:Bob		VicePresident-HumanResources:\$X	
VicePresident:*\$Y		VicePresident:*\$X	

4.1.4 Single Path Terms

A *single path term* is very similar to path expressions in XPath and XQuery. It is an expression of the following form:

$$p_1 T_1 p_2 T_2 \cdots p_n T_n \quad (4.3)$$

where $p_1, p_2, \dots, p_n$ are path separators, such as single forward slashes (/) and double forward slashes (//), and $T_1, T_2, \dots, T_{n-1}$ are element terms. For each $T_i = N : * T$ with $1 \leq i \leq n - 1$, N cannot be an attribute name and T cannot be a type, a value, a set of values, a set of object names or a set of ground or mixed object role terms. T_n is either an element term or a multiple path term, with $n \geq 1$. When T_n is a multiple path term, p_n must be omitted.

A single slash (/) indicates one single step in the path that is relative to the location preceding it in the tree structures of classes and objects. A double slash (//) indicates any number of steps in the tree structures of classes and objects, and it can be used to navigate all descendants of the context node preceding it. In other words, the double slash (//) does not indicate any number of steps in the graph of networked objects. However, we can use a path term that is a sequence of slashes and path terms to explore paths from one structure of an object to that of another object.

No cycle exists in the tree structures of classes and objects. All relationships are used to relate two different classes or objects in INM. In other words, there is no relationship that can link a descendant node to its ancestors in classes and objects. $p_i T_i$ can only be used to match a relationship expression in a class or an object, where T_i is element term of the form $N : * T, i < n$, and T must match a target node that denotes a reference of a class or an object. $p_i T_i$ stops at the target node and cannot explore into the real class or object. We can obtain the real class or object using T and use $p_{i+1} T_{i+1}$ to further match attribute expressions and various relationship expressions in the tree structure of the real class or object. Therefore, a single path term can always terminate appropriately. Additionally, a multiple path term contains multiple single path terms. A multiple path term can also terminate appropriately when all its single path terms terminate.

Example 3. The following single path terms can be used in schema queries:

/age:Int	/rank:\$Y	/\$X:Int
/VicePresident:Person	/\$X:Person	/\$X:Person/\$Y:\$Z
/VicePresident:\$X/\$Y:\$Z	/position:Hospital.\$Y/startYear:Int	
//VicePresident:\$X/gender:\$Y	//VicePresident-HumanResources:\$X	
/VicePresident[length:\$X, office:\$Y]	/VicePresident:Person[\$Y:String, age:Int]	
//Internist:\$X/worksIn:\$Y/status:\$Z/\$W:\$U		
/worksIn:Hospital/status:\$Y[D#:Int, takeCare:\$Z, manages:\$W]		
/VicePresident/VicePresident-HumanResources:\$X[age:Int, position:\$Y]		

The following single path terms can be used in instance queries:

/age:45	/rank:\$Y	/\$X:10
/*{Bob, Ben}	/\$X:*{Bob, Ben}	/\$X:Bob/\$Y:\$Z
/VicePresident:*\$X	/VicePresident:*{Bob, Ben}	
/VicePresident:*Bob/\$Y:\$Z	/worksIn:\$X/status:\$Y[D#:001, takeCare:\$Y]	
//VicePresident:*\$X[age:\$Y, gender:\$Z]	//VicePresident-HumanResources:\$X	
/VicePresident[length:3, \$X:A-501]	/position:OH.\$X	

```

/VicePresident:*Bob/$Y:$Z           //VicePresident:*$X[age:45,gender:$Y]
/worksIn:OH/status:Internist[D#:$X, takeCare:$Y, manages:$Z]
/VicePresident/VicePresident-HumanResources:$X[age:45, position:$Y]

```

4.1.5 Multiple Path Terms

A *multiple path term* is used to describe a multiple path with logical operators. It enables us to explore several different single paths at the same time. It is an expression of the following form:

$$[T_1 \ l_1 \ T_2 \ l_2 \ \cdots \ l_{n-1} \ T_n] \quad (4.4)$$

where $T_1, T_2, \dots, T_n$ are single path terms, and $l_1, l_2, \dots, l_{n-1}$ are logical operators, such as logical and (.) and logical or (|), with $n > 1$. For each T_i with $2 \leq i \leq n$, single slash (/) must be omitted if T_i starts with it. Logical and (.) has a higher priority than logical or (|).

For example, a multiple path term $[T_1, T_2|T_3, T_4|T_5]$ means $[(T_1, T_2)|(T_3, T_4)|T_5]$. We divide single path terms into groups by using logical or (|) as the separator. There are three groups: $\{T_1, T_2\}$, $\{T_3, T_4\}$, and $\{T_5\}$. The multiple path term is logically true if we can successfully explore at least one group of single path terms. Note that a variable can appear in several places in a multiple path term, and it holds the same value when it appears in the same group of single path terms. However, it is able to hold different values when it appears in different groups of single path terms.

Example 4. The following multiple path terms can be used in schema queries:

```

[$X:String, age:Int, position:$Y:$Z]      [D#:Int, takeCare:$Z, manages:$W]
[rank:$Y, $X:Person/$Y:$Z]                [rank:$Y, //VicePresident:$X/$Y:$Z]
[//VicePresident:Person[gender:$X, age:$Y] | $Z:Sickroom]
[//Internist:$X/worksIn:$Y/status:$Z/$W:$U | offers:Sickroom]

```

The following multiple path terms can be used in instance queries:

```

[$X:male, age:45, position:$Y]      [D#:001, takeCare:$Z, manages:$W]

```

$[rank:\$Y, //\$X:Bob/\$Y:\$Z]$ $[rank:\$Y, //VicePresident:*\$X/\$Y:\$Z]$
 $[//VicePresident:*\$Y/age:\$Z \mid rank:10]$
 $[//VicePresident:*\$Y[age:\$Z, gender:male] \mid rank:10]$
 $[//VicePresident:*\$Y[age:\$Z, gender:male] \mid rank:10]$
 $[age:43, worksIn:OH/status:Internist[D#:\$Y, takeCare:\$Z]].$
 $[//Internist:\$X/worksIn:\$Y/status:\$Z/\$W:\$U \mid offers:ICU-01]$

4.1.6 Class Terms

A *class term* is an expression of the following form:

$$C \ T \quad (4.5)$$

where C is either a class name or a variable, and optional T is either a single path term or a multiple path term.

Example 5. The following class terms are used in schema queries:

Hospital $\$Y$ $\$Y/age: \text{Int}$ Hospital/VicePresident:Person
 VicePresident/position: $\$X.\$Y/startYear: \text{Int}$
 $\$X/VicePresident/VicePresident-HumanResources[length:\$Y, office:\$Z]$
 Internist/worksIn:Hospital/status: $\$Y[D#: \text{Int}, takeCare:\$Z, manages:\$W]$
 Hospital[//VicePresident:Person[gender: $\$X$, age: $\$Y$] | $\$Z: \text{Sickroom}$]

4.1.7 ISA Terms

An *isa term* is used to retrieve superclasses or subclasses of a class. It is an expression of the following form:

$$C \text{ isa } C' \quad (4.6)$$

where C is either a variable or a class name, and C' is either a variable, a class name or a set of class names.

Example 6. The following isa terms are used in schema queries:

\$X isa \$Y \$X isa Person \$X isa {Doctor, Person}
 Internist isa \$Y Doctor isa Person Internist isa {Doctor, Person}

4.1.8 Object Terms

An *object term* is an expression of the following form:

$$O \ T \quad (4.7)$$

where O is either a variable or an object name and optional T is either a single path term or a multiple path term.

Example 7. The following object terms are used in instance queries:

OH \$X Bob/age:45 OH/*{Bob, Ben}
 \$R/worksIn:\$X/status:\$Y[D#:001, takeCare:\$Y]
 \$X/VicePresident/VicePresident-HumanResources:\$X[age:45, position:\$Y]
 OH[/VicePresident:*Bob[gender:\$X, age:\$Y] | \$Z:ICU-01]
 Jack[age:43, worksIn:OH/status:Internist[D#:\$Y, takeCare:\$Z]]

4.1.9 Classification Terms

An object can belong to object classes and role relationship classes. A *classification term* is used to find objects that belong to certain classes. It is an expression of the following form:

$$C \ O \quad (4.8)$$

where C is either a variable, a class name or a set of class names, and O is either a variable or an object name.

Example 8. The following classification terms are used in instance queries:

{VicePresident-MedicalAffairs, Patient} Ben
 {VicePresident-MedicalAffairs, Patient}\$X
 Hospital OH Person \$X

4.1.10 Schema Literals

A *schema literal* is used to express a subquery in a schema query. A class term or an isa term is a schema literal. Arithmetic, logical, comparison, string and set schema literals are defined using terms in an usual way. Class terms in Example 5 and isa terms in Example 6 are schema literals.

Example 9. The following schema literals are used in schema queries:

\$H=Hospital[/VicePresident:Person[gender:\$X, age:\$Y] | \$Z:Sickroom]
 \$X=String \$Y=Int

4.1.11 Instance Literals

A *instance literal* is used to express a subquery in an instance query. An object term $O\ T$ or a classification term $C\ O$ is an instance literal. $C\ O\ T$ is also an instance literal. Arithmetic, logical, comparison, string and set instance literals are defined using terms in an usual way. Object terms in Example 7 and classification terms in Example 8 are instance literals.

Example 10. The following instance literals are used in instance queries:

{VicePresident-MedicalAffairs, Patient}\$X=Ben/age:\$Y
 {VicePresident-MedicalAffairs, Patient}\$X[gender:male | worksIn:OH]
 Hospital \$X=OH[/VicePresident:*\$Y/age:\$Z | rank:10]
 Person \$X/worksIn:OH/status:Internist[D#:\$Y, takeCare:\$Z]
 \$Y > 30 \$Y=001 \$Z=Ann

Function Name	Description
count($\{ \$X \}$)	The number of values bound to $\$X$
avg($\{ \$X \}$)	The average of numbers bound to variable $\$X$
sum($\{ \$X \}$)	The sum of numbers bound to variable $\$X$
min($\{ \$X \}$)	The minimum of numbers bound to variable $\$X$
max($\{ \$X \}$)	The maximum of numbers bound to variable $\$X$

Table 4.1: Aggregation functions

4.2 Construction Terms

Construction terms are used to process query results in the user specified form.

4.2.1 Aggregation Terms

INM-QL supports built-in aggregation functions listed in Table 4.1, which are similar to other query languages, such as XQuery, SQL and so on. $\{ \$X \}$ is called a *set constructor* where $\$X$ is a variable. The set constructor constructs a set using all bindings for $\$X$. Functions *avg*, *sum*, *min* and *max* only accept a set of numeric values. However, *count* is able to accept any kind of values. All aggregation functions return a single numeric result.

An *aggregate term* is used to display a description and the result of an aggregate function. It is an expression of the following form:

$$d \ F \tag{4.9}$$

where d is a description string and F is an aggregation function.

Example 11. Given the following variable/value bindings:

$$\theta_1 = \{ \$Z/45 \}$$

$$\theta_2 = \{ \$Z/55 \},$$

the following aggregate term

“Average age of Vice-Presidents”:avg({\$Z})

constructs the following result:

Average age of Vice-Presidents:50

4.2.2 General Terms

A *general term* is used to display a description and information about objects in a specific order. It is an expression of the form:

$$d_1 X d_2 \text{ order by } T_1 t_1, \dots, T_n t_n \quad (4.10)$$

where optional d_1 and d_2 are description strings, X is a variable, $T_1, \dots, T_n$ are variables or aggregation functions and optional $t_1, \dots, t_n$ are sorting modifiers, such as *desc* and *asc*. *desc* denotes sorting in descending order and *asc* denotes sorting in ascending order, with $n \geq 0$. When t_i is omitted, order type comprises an ascending order by default. *order by* $T_1 t_1, \dots, T_n t_n$ is called *order-by part*, which is used to sort values bound to X by values bound to variables in $T_1, \dots, T_n$. It cannot be used in a schema query. Moreover, *order by* must be omitted if $n = 0$.

Example 12. Given the following variable/value bindings:

$\theta_1 = \{\$X/\text{OH}, \$Y/\text{Bob}, \$Z/45\}$

$\theta_2 = \{\$X/\text{OH}, \$Y/\text{Ben}, \$Z/55\},$

the following general term

VicePresident:\$Y order by \$Z desc

constructs the following result:

VicePresident:Ben

VicePresident:Bob

4.2.3 Path Terms

A *path term* is used to display values bound to variables along a path and dependency among them. It is an expression of the following form:

$$T_1/T_2/\cdots/T_n \quad (4.11)$$

where $T_1, \dots, T_{n-1}$ are general terms, T_n is a construction term with $n > 1$.

Example 13. Given the following variable/value bindings:

$\theta = \{\$X/\text{Hospital}, \$Y/\text{Person}, \$W/\text{Int}\},$

the following path term

$\$X/\text{VicePresident-HumanResources}:\$Y/\text{age}:\$W$

constructs the following result:

Hospital
VicePresident-HumanResources:Person
age:Int

Example 14. Given the same variable/value bindings:

$\theta_1 = \{\$X/\text{OH}, \$Y/\text{Jack}, \$Z/43, \$W/\text{male}\}$

$\theta_2 = \{\$X/\text{OH}, \$Y/\text{Jay}, \$Z/52, \$W/\text{male}\},$

the following path term

$\text{Hospital}:\$X/\text{"number of doctors":count}(\{\$Y\})$

constructs the following result:

Hospital:OH
number of doctors:2

Example 15. Given the same variable/value bindings as in Example 14, the following path term:

$\text{Hospital}:\$X/\{\text{Doctor}:\$Y \text{ order by } \$Z\}$

constructs the following result:

```
Hospital:OH
  Doctor:Jack
  Doctor:Jay
```

4.2.4 Tuple Terms

A *tuple term* is used to display values bound to variables in a multiple path and dependency among them. It is an expression of the following form:

$$T [T_1, \dots, T_n] \quad (4.12)$$

where T is a general term, $T_1, \dots, T_n$ are construction terms with $n \leq 0$. In general term T , X must bind to either an object name or a class name. When $n = 0$, the tuple term displays information of the entire object or class.

Example 16. Given the following variable/value bindings:

$\theta = \{\$X/\text{Hospital}, \$Y/\text{Person}\},$

the tuple term `Definition:$Y[ ]` constructs the following result:

```
Definition:Person[age:Int, gender:String]
```

Example 17. Given the following variable/value bindings:

$\theta = \{\$X/\text{Hospital}, \$Y:\text{Person}, \$W:\text{Int}, \$Z:\text{Sickroom}\},$

the following tuple term

`$X["VicePresident":$Y/age:$W, offers:$Z]`

constructs the following result:

```
Hospital[
  VicePresident:Person
  age:Int,
  offers:Sickroom]
```

Example 18. Given the same variable/value bindings as in Example 12, the following tuple term:

```
Hospital:$X["Number of Vice-President":count({$Y}),
            "Average age of Vice-Presidents":avg({$Z}),
            "Maximum age of Vice-Presidents":max({$Z}),
            "Minimum age of Vice-Presidents":min({$Z})]
```

constructs the following result:

```
Hospital:OH[
    Number of Vice-President:2,
    Average age of Vice-Presidents:50,
    Maximum age of Vice-Presidents:55,
    Minimum age of Vice-Presidents:45]
```

4.2.5 List Terms

A *list term* is used to individually display query results for different construction terms. It is an expression of the following term:

$$T_1, \dots, T_n \quad (4.13)$$

where $T_1, \dots, T_n$ are construction terms with $n > 1$.

Example 19. Given the same variable/value bindings as in Example 16, the following list term:

```
Name:$X, Definition:$Y[ ]
```

constructs the following result:

```
Name:Hospital,
Definition:Person[age:Int, gender:String]
```

Example 20. Given the same variable/value bindings as in Example 17, the following list term:

\$Z, \$X[VicePresident:\$Y/age:\$W, offers:\$Z]

constructs the following result:

Sickroom,
Hospital[
 VicePresident:Person
 age:Int,
 offers:Sickroom].

Example 21. Given the same variable/value bindings as in Example 14, the following list term:

Hospital:\$X/Doctors:{\$Y/age:\$Z}, "average age of doctors":avg({\$Z})

constructs the following result:

Hospital:OH
 Doctors:
 Jack
 age:43
 Jay
 age:52,
 average age of doctors:45.25

4.2.6 Grouping Terms

A grouping term is used to display a description and the grouping results. It is an expression of the following form:

$$d \{ T \} \tag{4.14}$$

where optional d is a description string, T is a construction term that is not an aggregation term. The following are grouping terms:

$\{\$W\}$ $\{\$Y/\text{age}:\$Z\}$ $\{\$Y[\text{age}:\$Z, \text{gender}:\$W]\}$

which used in the above Example 22, Example 23, and Example 24 respectively.

Example 22. Given the following variable/value bindings:

$\theta = \{\$X/\text{Jack}, \$Y/\text{OH}, \$Z/001, \$W/\text{Ann}\},$

the following path term

$\$X/\text{worksIn}:\$Y[\text{D\#}:\$Z, \text{takeCare}:\{\$W\}]$

constructs the following result:

```

Jack
  worksIn:OH[
    D#:001,
    takeCare:Ann]

```

Example 23. Given the same variable/value bindings as in Example 14, the following path term:

$\$X/\text{Doctors}:\{\$Y/\text{age}:\$Z\}$

constructs the following result:

```

OH
  Doctors:
    Jack
      age:43
    Jay
      age:52

```

Example 24. Given the same variable/value bindings as in Example 14, the following path term:

$\$X/\text{Doctors}:\{\$Y[\text{age}:\$Z, \text{gender}:\$W]\}$

constructs the following result:

OH

Doctors:

```

    Jack[
      age:43,
      gender:male]
    Jay[
      age:52,
      gender:male]

```

4.2.7 Pair Terms

A *pair term* is used to display an expression of an attribute, a relationship or an identification. It is an expression of the form:

$$X : T \quad (4.15)$$

where X is a variable, T is either a variable or a grouping term.

Example 25. Given the same variable/value bindings:

```

 $\theta_1 = \{\$X/\text{Bob}, \$Y:\text{age}, \$Z:45\}$ 
 $\theta_2 = \{\$X/\text{Bob}, \$Y:\text{gender}, \$Z:\text{male}\}$ 
 $\theta_3 = \{\$X/\text{Bob}, \$Y:\text{position}, \$Z:\text{OH.VicePresident-HumanResources}\}$ 
 $\theta_4 = \{\$X/\text{Ben}, \$Y:\text{age}, \$Z:55\}$ 
 $\theta_5 = \{\$X/\text{Ben}, \$Y:\text{gender}, \$Z:\text{male}\}$ 
 $\theta_6 = \{\$X/\text{Ben}, \$Y:\text{position}, \$Z:\text{OH.VicePresident-MedicalAffairs}\}$ 
 $\theta_7 = \{\$X/\text{Ben}, \$Y:\text{health}, \$Z:\text{OH.Patient}\},$ 

```

the following path term that contains the pair term $\$Y:\Z :

```
VicePresident:$X/{\$Y:$Z}
```

constructs the following result:

VicePresident:

Bob

age:45

gender:male

position:OH.VicePresident-HumanResources

VicePresident:

Ben

age:55

gender:male

position:OH.VicePresident-MedicalAffairs

health:OH.Patient

4.3 Query

A query consists of two parts: a query expression and a result construction expression. The query expression is used to find variable/value bindings. The result construction expression is used to process variable/value bindings in the user specified form.

4.3.1 Schema Queries

A schema query is an expression of the following form:

$$\text{query class } L_1, L_2, \dots, L_n \text{ construct } R \quad (4.16)$$

where $L_1, L_2, \dots, L_n$ are schema literals, with $n > 0$; R is a construction term.

Query 5. To retrieve class *Hospital*, target classes of its relationships *VicePresident* and *offers*, and the data type of attribute *age* in target class of *VicePresident*, and display query result information in a tuple form, we can use the following schema query:


```
query class $X=Hospital[/VicePresident:$Y, offers:$Z], $Y/age:$W
construct $X[VicePresident:$Y/age:$W, offers:$Z]
```

Query 5 has two schema literals: $\$X=Hospital[/VicePresident:\$Y, offers:\$Z]$ and $\$Y/age:\W . The first schema literal consists of a comparison term $\$X=Hospital$ and a multiple path term $[/VicePresident:\$Y, offers:\$Z]$. The comparison term binds variable $\$X$ to class *Hospital*. The multiple path term retrieves target classes of relationships *VicePresident* and *offers* at the same time in the class *Hospital*, and binds variables $\$Y$ and $\$Z$ to those two target classes. The second class term consists of a variable $\$Y$ and a single path term that has an element term $/age:\$W$, and it retrieves the data type of attribute *age* in target class bound to variable $\$Y$. Moreover, Query 5 retrieves the same variable/value bindings and has the same tuple term as the Example 17; hence, its result construction expression constructs the same result.

Query 6. To retrieve classes that have attribute *rank* whose data type is *Int* and relationship *VicePresident*, and also retrieve *VicePresident*'s target class that has an attribute *age* with data type *Int*, and display query result in a list form, we can use the following schema query:

```
query class $X[rank:Int, VicePresident:$Y], $Y/age:Int
construct Name:$X, Definition:$Y[]
```

Query 6 has two schema literals: $\$X[rank:Int, VicePresident:\$Y]$ and $\$Y/age:Int$. The first schema literal consists of a variable and a multiple path term. The variable binds to satisfying classes. The multiple path term matches classes that have attribute *rank* with data type *Int* and relationship *VicePresident*, retrieves the target class of *VicePresident*, and binds variable $\$Y$ to target classes. The second class term consists of a variable $\$Y$ and a single path term, and it explores target classes bound to variable $\$Y$ to match attribute *age* with data type *Int*. Moreover, Query 6 retrieves the same variable/value bindings and has the same list term as the Example 19; hence, its result construction expression constructs

the same result.

Query 7. To retrieve classes that have relationship *VicePresident-HumanResources*, target class of *VicePresident-HumanResources*, retrieve data type of attribute *age* in the target classes, and display query results in a path form, we can use the following schema query:

query class \$X//VicePresident-HumanResources:\$Y/age:\$W
construct \$X/VicePresident-Human Resources:\$Y/age:\$W

Query 7 has a schema literal that consists of a variable \$X and a single path term //VicePresident-HumanResources:\$Y/age:\$W. The variable \$X binds to satisfying classes. The single path term consists of two element terms: //VicePresident-HumanResources”:\$Y and /age:\$W. The first element term explores classes that have relationship *VicePresident-HumanResources*, retrieves target class of *VicePresident-HumanResources* and binds variable \$Y to the target class. The second element term continues on retrieving data type of attribute *age* in the target class bound to variable \$Y. Moreover, Query 7 retrieves the same variable/value bindings and has the same path term as the Example 13 and returns the same result.

4.3.2 Instance Queries

An instance query is an expression of the following form:

$$query L_1, L_2..., L_n construct R \quad (4.17)$$

where $L_1, L_2..., L_n$ are instance literals with $n > 0$; R a construction term.

Query 8. To retrieve a hospital *OH*, number of Vice-Presidents in hospital *OH*, and average, maximum, and minimum of Vice-Presidents’ ages, we can use the following instance query:

query Hospital \$X=OH//VicePresident:*\$Y, \$Y/age:\$Z

```

construct Hospital:$X["Number of Vice-President":count({$Y}),
    "Average age of Vice-Presidents":avg({$Z}),
    "Maximum age of Vice-Presidents":max({$Z}),
    "Minimum age of Vice-Presidents":min({$Z})]]

```

Query 8 has two instance literals: $\$X=OH//VicePresident:*\Y and $\$Y/age:\Z . The first instance literal consists of a class name, a comparison term, and a single path term that has one path element term. The comparison term $\$X=OH$ binds variable $\$X$ to object OH that belongs to class *Hospital*. The element term $//VicePresident:*\$Y$ retrieves targets of role relationship *VicePresident* and all targets of its role sub-relationships in object OH , and binds variable $\$Y$ to obtained targets. The second instance literal continues on retrieving values of attribute *age* in targets bound to variable $\$Y$. Moreover, Query 8 retrieves the same variable/value bindings and has the same object property construction term as the Example 18 and returns the same result.

Query 9. To retrieve all vice-presidents in hospital OH that has a relationship *VicePresident* or an attribute *rank* with the value of 10 and to sort vice-presidents by their ages in ascending order, we can use the following instance query:

```

query Hospital OH[//VicePresident:*\$Y/age:\$Z | rank:10]
construct VicePresident:\$Y order by \$Z desc

```

Query 9 has an instance literal $Hospital\ OH[//VicePresident:*\$Y/age:\$Z\ |\ rank:10]$. The instance literal consists of a class name, an object name, and a multiple path term. The object OH belongs to class *Hospital*. The multiple path term has two single path terms separated by a logic or ($|$). The first single path term consists of two element terms: $//VicePresident:*\$Y$ and $/age:\$Z$. The first element term retrieves targets of role relationship *VicePresident* and all targets of its role sub-relationships in object OH , and binds targets to variable $\$Y$. The second element term retrieves values of attributes *age* in targets bound to variable $\$Y$. The second single path term matches a attribute *rank* with the value of 10

in object *OH*. The multiple path term means that object *OH* must have a role relationship *VicePresident* whose targets have attributes *age* or an attribute *rank* with the value of 10. Query 9 retrieves the same variable/value bindings and has the same object property construction term as the Example 12 and returns the same result.

Query 10. To retrieve persons who are 43 years old and who work in *OH* as internists, to retrieve context dependent information about their doctor ID and patients that they take care of, and to display query results in a tuple form, we can use the following instance query:

```
query Person $X[age:43, worksIn:OH/status:Internist[D#:$Y, takeCare:$Z]]
construct $X[$Y, {$Z}]
```

Query 10 has an instance literal that has a variable and a multiple path term. The variable binds satisfying objects that belong to class *Person*. The multiple path term has two single path terms: *age:43* and *worksIn:OH/status:Internist[D#:\$Y, takeCare:\$Z]*. The first single path term has an element term that means the satisfying object must have an attribute *age* with the value of 43. The second single path term has two element terms: *worksIn:OH*, */status:Internist*, and a multiple path term *[D#:\$Y, takeCare:\$Z]*. The first element term matches a context relationship *worksIn* with a context relationship target node *OH* in a satisfying object. The second element term matches an identification *status* with a context relationship node *Internist* under the context relationship target node *OH*. The last multiple path term retrieves value of attribute *D#* and targets of relationship *takeCare* under context relationship node *Internist*. Moreover, Query 10 retrieves the same variable/value bindings and has the same tuple construction term as the Example 22 and returns the same result.

4.4 Features of INM-QL

We summarize the following features of INM-QL:

- INM-QL consists of schema queries and instance queries. Schema queries are used to explore networked classes at the schema level and to retrieve data of classes, their attribute, relationships, subclasses and context-dependent information. Instance queries are designed to explore networked objects at the instance level and to retrieve data of objects, their attributes, relationships and context-dependent information.
- Variables are all logical variables that are place holders and have no types. They can be used to bind to anything of objects based on its locations in the query, which make INM-QL flexible and easy to use in practice.
- INM-QL supports single and multiple path expressions to explore networked classes and objects at the schema level and at the instance level.
- Attributes and various relationships are treated in the same way. The system automatically figures out which are attributes, which are relationships, and their kinds; hence, the user is not required to explicitly specify attributes and various relationships.
- INM-QL provides result construction expression to process query results in the user specified form. It also supports operations, such as order by, aggregate and grouping functions, which are integrated into result construction expression.

Chapter 5

Implementation

This chapter describes the implementation of INM-QL. It begins with an overview of the architecture of INM database management system, and introduces its storage, index mechanism, query processing, and evaluation strategies. It then describes the user interfaces and shows some experimental results.

The INM database management system has been implemented in C using GLib (version 2.22.3) [43] and Berkeley DB (version 4.7) [38]. A lexical and syntactic analyzer that supports INM-DDL, INM-DML and INM-QL has been implemented using Flex (version 2.5) and Bison (version 1.25) [28].

GLib is a utility library that simplifies programming in C. It addresses three fundamental problematic issues of C: data containers, portability and utility. It provides many useful data types, macros, type conversions, string utilities and so on. It also provides containers such as linked list, queue, hash table and tree ; it has build-in routines for those data structures. It is very useful in writing portable code, because it works on many platforms such as Unix/Linux, Windows, OS/2, and BeOS.

Berkeley DB is an open source and general-purpose library that provides a high-performance embedded database. It supports ACID transactions and recovery, and concurrent access by

multiple users. It is not a full-fledged database management system that supports data definition, data manipulation and data retrieval. It can be linked directly to an application, so that the application can make simple function calls, rather than send requests to the remote server. As a result, Berkeley DB eliminates the overhead of processing the query languages. It is very flexible and controllable for developers to configure and optimize for specific applications. It also offers an in-memory cache for rapid access to frequently used data, in order to further improve its performance.

Each database record in Berkeley DB contains two parts: a key and some arbitrary data. Both key and data are byte arrays. Complex structures can be stored in the database, as long as they reside in a single contiguous block of memory.

Berkeley DB has two kinds of databases: primary databases and secondary databases. A primary database is used to store data, whereas a secondary database provides an alternative key to access that data. In a secondary database, the keys are alternative keys that correspond to data or a part of data in a primary database, and the values are keys in that primary database.

5.1 System Overview

The INM database management system employs a conventional thin client/fat server architecture, as shown in Figure 5.1.

Client The client of the database management system is organized into two layers: the graphical interface and the communication module. The graphical interface sends user requests to the server via the communication module and takes the query results from the server for proper display. The second layer is the communication module. It sends user requests from the first layer to the server for parsing and execution, and obtains query results from the server and then sends them back to the first layer.

Figure 5.1: System Architecture

Server The server is organized into six layers. The first layer is the communication module. It communicates with the communication module from the client's side and obtains user requests, passes them to the query manager, and then ships the query results back to the client.

The second layer is the scheduler. It obtains multiple user requests from the communication module, schedules user requests, and ships them to the lexical and syntactical parser.

The third layer is the lexical and syntactical analyzer that is implemented by using Bison and Flex. It performs lexical and syntactical analysis of user requests. It filters out invalid requests, transforms strings of valid requests into standard forms and sends them to the query manager.

The DDL/DML and query managers of the forth layer are responsible for DDL/DML and query processing. The DDL/DML manager validates operations and checks various integrity constrains. The query manager decides on what evaluation strategies to use according to the nature of the query, which is discussed in Section 5.6. It invokes the schema and the instance manager at the next layer to handle queries of classes and objects respectively. It also keeps track of what is in main memory and removes what should be freed from the main memory after finishing the query processing and sending the query results to clients.

The fifth layer consists of class and object managers. It is in charge of storing, retrieving, modifying and deleting classes and objects in databases.

The last layer is the storage manager. It is in charge of the management of disk-based data structures and loading data into main memory from disk as needed. It is implemented in Berkeley DB to provide rapid access to classes, objects, and other meta information about them on the disk.

5.2 Storage

Berkeley DB is used to store the classes, objects and meta information about them. There are six primary files: class file, object file, class-attribute-relationship file, object-attribute-relationship file, inheritance file and class-object file. Indexes are automatically mentioned on class file, object file, class-attribute-relationship file and object-attribute-relationship file to provide alternative ways to access classes, objects, and attributes and relationships at the schema level and at the instance level. In Section 5.6, we describe how to use index in the query processing. The structures of the primary files and index files are shown in Figure 5.2.

We convert tree structures of classes and objects shown in Section 3.2 and Section 3.4 to byte array as values of records in class file and object file. All of the relevant data of classes and objects is physically and adjacently stored in the database. This kind of storage enables us to use a few I/O operations to get whole data structure of class and object, and to avoid wasting time on the join operation. Oracle also uses the same idea to reduce disk operation I/O. It uses cluster and cluster index [17] to combine rows from one or more tables and to store them in the same block. XML also uses cluster and clustering storage to improve retrieving data from the disk and to increase the speed of query [48].

Class File All classes are stored in the class file. Each class has an ID generated automatically by the system as the key for each class, and the value of each record is a byte array of a class's tree structure. This byte array consists of ID, class name, attributes and various relationships. The system automatically generates unique IDs for attributes, relationships and identifications of classes. Each attribute is composed of its ID, name and type, which has the format attribute:type. Each relationship consists of its ID and name, ID and name of target class, attributes, relationships, and context-dependent properties. Each identification is composed of its ID and name, and ID and name of a role relationship in another class and

Figure 5.2: Database Structures

Figure 5.3: Part of Class file Example

its context-dependent properties. The class file has a secondary index file whose key column stores class names and value column stores IDs of corresponding classes. Figure 5.3 illustrates partial internal structures of classes *Hospital*, *Person*, *Sickroom*, and *Internist* from Figure 3.5, Figure 3.3 and Figure 3.6, and it demonstrates index on the class file.

Figure 5.4 further illustrates the structure of class *Hospital* in some detail, including how the regular relationship *offers*, the hierarchy of role relationship *VicePresident*, and their target classes are stored. Figure 5.5 further demonstrates the structure of class *Internist* in some detail, including how context relationship information is stored. The segment of context relationship *worksIn* of class *Internist* maintains context-dependent information and the segment of identification target *Internist* stores the name and ID of a role relationship *Internist* in the class *Hospital*, and also stores nested context-dependent properties.

Figure 5.6: Part of Object File Example

Object File All information about objects is stored in object file. Each object has an ID that is generated automatically by the system as the key for each object; the values of each record are a byte array of the object structure. This byte array contains the ID, object name, name of its class, attributes, various relationships and targets of relationships. The system automatically generates unique IDs for attributes, relationships and object identifications. Each attribute is composed of its ID, name, and value. Each relationship consists of its ID and name, the names and IDs of target objects, attributes, relationships and context-dependent properties. Each identification is composed of its name and ID, the ID and name of a role relationship in another object and context-dependent properties. The object file has a secondary index file whose key column stores object names and value column stores sets of IDs of corresponding objects. Figure 5.6 illustrates the internal structures of objects *OH*, *Bob*, *Ben*, *Ann*, *Jack*, and *Jay* from Figure 3.7 and Figure 3.8, and the secondary index on the object file.

Figure 5.7 further illustrates the structure of object *OH* in some detail, including the hierarchy of role relationship *VicePresident*, the hierarchy of role relationship *Doctor* and

Figure 5.7: Role Relationship Structure of Object *OH*

their targets. Figure 5.8(a) further demonstrates the structure of object *Jack* and shows how context-dependent information is stored. The segment of context relationship *worksIn* in object *Jack* maintains context-dependent information and the segment of identification target *Internist* stores the name and ID of a role relationship *Internist* in the object *OH* and nested context-dependent properties: a context-dependent attribute *D#* with the value of *001* and two context-dependent relationships *manages* with the target of *ICU-01* and *take-Care* with the target of *Ann*. Figure 5.8(b) further demonstrates the structure of object *Jay*. The segment of context relationship *worksIn* in object *Jay* maintains context-dependent information, and the segment of identification target *Oncologist* stores the name and ID of a role relationship *Internist* in the object *OH* and a nested context-dependent attribute *D#* with the value of *002*.

Figure 5.8: Context-Dependent Information Structures of Objects *Jack* and *Jay*

Object-Attribute-Relationship File The object-attribute-relationship file stores some redundant information about the objects' attributes, relationships and identifications to build the index file that serves for query processing. If we want to update and delete classes and objects, we could use IDs to find corresponding attributes, relationships and identifications of objects, and then update and delete them in the file.

The system automatically inserts information about the objects' attributes, relationships and identifications into an object-attribute-relationship file, when objects are inserted. Key column stores IDs of objects' attributes, relationships and identifications. For attributes, value column stores byte arrays that contains IDs and names of attributes, values, and instance paths of attributes. For relationships or identifications, it stores byte arrays that contain IDs and names of relationships or identifications, names and IDs of targets, and instance paths of relationships or identifications. An instance path is a list of pairs of names and IDs, where names can be either object names, role relationship names, target

names of context relationships or target names of identifications. IDs can belong to either objects, role relationships, targets of context relationships or targets of identifications. It shows a path starting from the root of a tree structure of an object to access an attribute, a relationship or an identification.

The object-attribute-relationship file has a secondary index file. Its key column stores either attribute names, values, combinations of attributes' names and their values, relationship names, target names, combinations of relationship names and their target names, identification names, target names of identifications, or combinations of identification names and their target names. For an attribute of the form *attribute:value*, the key can be *attribute*, *:value*, and *attribut:value*. For a relationship of the form *relationship:target*, the key can be *relationship*, *:target*, *relationship:target*. For a relationship of the form *identification:target*, the key can be *identification*, *:target*, *identification:target*. Its value column stores sets of IDs of corresponding attributes, relationships or identifications in object-attribute-relationship file. With very little information about attributes, relationships and identifications, we can easily access them in class-attribute-relationship file. With an instance path in an object-attribute-relationship record, we can easily obtain an object to which a corresponding attribute, relationship or identification belongs.

Figure 5.9 shows how the object-attribute-relationship file maintains instance paths of several attributes and relationships *rank*, *age*, *VicePresident*, *VicePresident-HumanResources*, *Doctor* and *worksIn*. For example, the instance path of attribute *age* is $((OH, 1))$; that means attribute *age* is stored in class-object node *OH*. The instance path of role relationship *VicePresident-HumanResources* is $((OH, 1), (VicePresident, 19))$; that means we can travel through class-object node *OH* and role relationship node *VicePresident* in the tree structure of object *OH* and access the role relationship node *VicePresident-HumanResources*. Moreover, there are two context relationships *worksIn* in object-attribute-relationship file. These two *worksIn* have different IDs and two different instance paths, which means they belong to two objects: *Jack* and *Jay*. Figure 5.9 also demonstrates how we build index on

Figure 5.9: Object-Attribute-Relationship File and Its Index

the object-attribute-relationship file. For example, to find an attribute *rank* with the value of *10*, we can use search key *rank:10* and get one *rank* in object *OH*. To find a context relationship *worksIn* with target *OH*, we can use search key *worksIn:OH* and get two *worksIn* in two different objects *Jack* and *Jay*. To find an identification *status*, we can use search key *status* and get two identifications in the two different objects *Jack* and *Jay*. To find an identification *status* with target *Internist*, we can use search key *status:Internist* and get an identification in object *Jack*. Hence, it is more likely to obtain much less and more precise results if we provide more information when searching through the index file.

Class-Object File				
Key	Value			
1	1	Hospital	...	1
...	...			
4	4	Patient	...	9 10
5	5	VicePresident-HumanResources	...	8
6	6	VicePresident-MedicalAffairs	...	9
...	...			
11	11	Internist	...	11
12	11	Oncologist	...	12

Figure 5.10: Part of Class-Object File Example

Class-Attribute-Relationship File We use the same idea as the object-attribute-relationship file for class to provide another rapid way to access classes in the database. The class-attribute-relationship file stores core information of classes' attributes, relationships and identifications. It also has a index file like the object-attribute-relationship file.

Inheritance File The inheritance file stores information about subclasses and super classes for classes. For each file record, there are two columns: key and value. The key column stores class IDs and the value column stores byte arrays that contain names, IDs and types of classes, IDs of their subclasses, and IDs of their super classes.

Class-Object File The class-object file stores the correspondence between classes and objects that belong to those classes. For each file record, there are two columns: key and value. The key column stores class IDs and the value column stores byte arrays that contain names, IDs and types of classes, and IDs of their objects. Figure 5.6 illustrates sample data in the class-object file.

Note that all following query examples are based on data shown in Figure 5.6 and Figure 5.3.

5.3 Result Tree

Before discussing about query processing, we need to have an intermediate data structure to store variable/value bindings.

A result tree is a tree of variable/value binding, which is used to store variable/value bindings and dependence among variables. Every tree has a root that stores no information about the variable/value binding. Every node except the root in a tree stores a variable and one value bound to the variable. The value can be anything of a class or an object, such as a number, a string, a memory address of a class or an object, an attribute, and a relationship. Note that values of a variable are dependent on the value of the variable that appears in the previous path element term in a path expression. One node can have one edge linking to its parent, which shows that variable/value binding in it is dependent on its parent's.

Consider the following query expression:

query Hospital \$X/Doctor:*\$Y /worksIn:\$Z

we can obtain following variable/value bindings:

$$\theta_1 = \{\$X/OH, \$Y/Jack, \$Z/OH\},$$

$$\theta_2 = \{\$X/OH, \$Y/Jack, \$Z/OH\},$$

where values of the variable \$Y are objects *Jack* and *Jay* when value of variable \$X is object *OH*; variable \$Z binds to *OH* when variable \$Y binds to object *Jack*; variable \$Z binds to *OH* when variable \$Y binds to object *OH*. We build a result tree shown in Figure 5.11 for the above query expression.

Hash Table of Variable/Values The tree of variable/value bindings alone is not enough to represent intermediate query result, as it contains no information about the locations of the variables. A variable can scatter everywhere in the tree, which leads to traverse the entire tree for every variable. Traversing the entire tree many times is very expensive and inefficient when the tree has a large number of nodes with several levels.

The hash table of variable/values is used to solve the above problem. It is very useful to

Figure 5.11: Structures of result tree

locate a variable and its values in the tree in a constant time. The key is a name of a variable and the value is a list of memory addresses of nodes that belong to that variable in the result tree. Having built a result tree for a subquery, a hash table of variable/values is built to help process following subqueries and handling output. Note that it is still required to traverse a whole tree of variable/value binding once to build a hash table of variable/values. With the hash table of variable/values, we do not have to traverse the tree a number of times when subqueries require values of variables and construct clause asks to output values of variables. One example of the hash table of variable/values is shown in Figure 5.11.

Hash Table of Object and Hash Table of Class The same object or class can be values of different variables and can scatter everywhere in the tree. If the system does not track what have been loaded to main memory, the query manager treats every object or class as a new one and loads it from database with its ID. As a result, a number of unnecessary I/O would be occurred and a lot of duplicated objects and classes would occupy a large chunk of memory. The query processing takes a lot of time to read disk and wastes space in the main memory, which is very expensive and inefficient.

The hash tables of objects and classes are used to manage classes and objects loaded into the main memory. For each query request, the hash tables of objects and classes are generated when the query manager begins to process the query request. The keys of the hash table of object and the hash table of class are the ID of object and the ID of the class respectively, the values of these two hash table are the corresponding object and class respectively. When the query manager wants to load an object and bind one variable to the object, it will always check whether the object is in the hash table of object. If the object is in, that means the object was already loaded in the main memory before, so it will not be necessary to load the object into main memory again; the variable will directly bind to the memory address of the object. If the object is not in, then it will be loaded into the main memory, the query manager will add it in the hash table of object and will bind the variable to it. The query manager will perform the same procedure when it wants to load a class. Thus, values of different variables that bind to the same objects or classes will always share the same memory addresses in the result tree. One example of the hash table of object is shown in Figure 5.11.

5.4 Query Processing

The query manager handles queries and uses the meta information in the query statements to decide which classes and objects should be loaded into the main memory. It also keeps track of what is in main memory and removes what should be freed from the main memory after finishing query processing and sending query results to clients.

There are two main procedures to process a query: parser and execution. The processing is illustrated in Figure 5.12.

We build a bottom-up parser by using Flex and Bison. The parser consists of a lexical analyzer and a semantic analyzer. Flex [16] is used to build the lexical analyzer. The lexical analyzer takes a query string that is a sequence of characters and breaks them into tokens.

Figure 5.12: Query Processing

Bison [32] is used to build our semantic analyzer and it converts an annotated context-free grammar into an LALR(1) parser. The semantic analyzer takes generated sequential tokens from the lexical analyzer to check against grammar, acts differently according to different grammar statements, and generates a complex tree structure for presenting the query string.

The query manager is responsible for further processing the query tree structure. It consists of following components: query optimization, return pattern extraction, result tree generation and query output handler. Figure 5.13 illustrates how all of these components work together. The query manager takes a query tree structure from the lexical and syntactic analyzer as an input and generates a output string as query results. The query manager first chooses a evaluation strategy for a query expression in query optimization and builds a result tree to hold variable/value bindings. It simultaneously extracts variables from result construction expression to make sure that the variables appear in the query expression. Finally, the query output handler uses the result tree to generate query results in term of

Figure 5.13: Query Manager

return pattern. It supports many construction terms to form query results. And many different algorithms are used to deal with different construction terms. We discuss the query optimization and the query output handler in Section 5.6 and Section 5.7.

Return Pattern Extraction The return pattern extraction is a special processing procedure in query execution before handling query output. It extracts variables from the result construction expression in query tree structure to check if there is a variable that is not used in the query expression. If the variable cannot be found in query expression, it is impossible to get the variable/value bindings for this variable. If so, it raises an error and returns the error to the user before getting into executing query expression. It also preprocesses the construct clause so as to make construct clause convenient to execute for query output handler.

Result Tree Construction The result tree construction is a core procedure to build a result tree for a query expression to hold variable/value bindings and dependency among them, to maintain hash tables of classes and objects, and to traverse the result tree to map

each variable to a set of nodes that contains its values, which is stored in hash table of variable/values.

One core procedure of result tree construction is to merge two or more result trees into a completed result tree. The query manager first takes two result trees R_1 and R_2 . For each node N_2 at the second level of R_2 , if the query manager finds a node N_1 that is at the second level of R_1 and contains the same variable/value bindings as N_2 , the query manager adds subtrees of N_2 to N_1 as subtrees; otherwise, it adds the subtree whose root is N_2 under the empty root of R_1 . The query manager continues on merging rest of result trees to R_1 and finally constructs a completed result tree that is stored in R_1 .

5.5 Path Element

Before discussing about query optimization, we need to introduce the path element, which is the smallest element in a path expression. It is used to match either an attribute, a regular relationship, a role relationship, a context relationship or an identification in a tree structure of an object or a class, and bind variables to corresponding values based on their locations. In a single path term of the following form $p_1 T_1 p_2 T_2 \cdots p_n T_n$, each $p_i T_i$, where T_i is an element term with $1 \leq i \leq n$, is called a *path element*. An element term has the form of $n : * T$.

Example 26. Let A, B, C, D, E denote arbitrary strings, and \$X, \$Y, and \$Z variables, the following examples show typical cases of path elements:

(1) //A:{B, C}	/A:#{B, C}	/A:B.C	/A:{B.C, D.E}	
(2) /\$X	//\$X	/*\$X	/\$X:\$Y	//\$X:*\$Y
(3) //:#{B, C}	//\$X:B	/\$X:*B	/\$X:#{B, C}	
/A:\$X	/A:*\$X	//:{B.C, D.E}	/\$X:{B.C, D.E}	
//A:B.\$Y	//A:\$X.B	//:B.\$X	/{B.\$X, \$Y.D}	
//\$X:{B.\$Y, D.E}	//\$X:B.\$Y			

Different algorithms are used to process different cases of path elements to match attributes, relationships, and identifications, and to bind variables to values.

Before discussing how to match, we further classify path elements into three groups based on their amount of information: simple cases, normal cases and general cases, when we execute the query expression. See Section 5.6.6 for more details.

Simple Cases A simple case has no variables, in which n can be a name including attribute name, relationship name and identification name, T can be either a type, a class name, a value, an object name, a ground object role term, a set of values, a set of object names or a set of ground object role terms. Typical examples of simple cases are listed in Example 26 (1).

General Cases A general case always contains nothing but variables. That is, n and T can only be variables, and n and T can be omitted, but not at the same time. Typical examples of general cases are listed in Example 26 (2):

Normal Cases The rest of path elements that are neither simple cases nor general cases are normal cases. Typical examples of normal cases are listed in Example 26 (3).

A path element needs a starting node, a node of tree structure of a class or an object to start matching. INM-QL does not explicitly differentiate attributes, various relationships and identifications in path elements. As a result, the query manager must explore all attributes, regular relationships, role relationships, context relationships and identifications in the starting node and its children, and try to match the path element with one of them. However, some operation signs of path elements can be helpful to quickly determine with what the path elements match, such as wildcard (*) and dot (.). If a path element contains a wildcard (*), the query manager only matches it with role relationships whose targets or role sub-relationships's targets satisfy target names. If a path element contains a dot (.),

then the query manager only matches it with identifications whose targets satisfy its object role term.

The basic idea of a path element matching is to match all its non-variables with either an attribute, a relationship or an identification, and then bind variables to corresponding values based on their locations. If a path element contains a wildcard (*), we can present data about an arbitrary role relationship in the database in the following form:

$$r : * \{o_1, \dots, o_n\}$$

where r is the name of the role relationship, $o_1, \dots, o_n$ are its targets and targets of its role sub-relationships.

The path element $/A:* \{B, C\}$ matches the above role relationship if A is the same as r and $\{B, C\}$ is a subset of $\{o_1, \dots, o_n\}$. The path element $//A:* \$X$ matches the above role relationship if A is the same as r and variable $\$X$ binds o_i with $1 \leq i \leq n$. The path element $\$X:* \{B, C\}$ matches the above role relationship if $\{B, C\}$ is a subset of $\{o_1, \dots, o_n\}$, and variable $\$X$ binds to r . The path element $/\$X:* \Y matches the above role relationship with no condition, variable $\$X$ binds to r and variable $\$Y$ binds o_i with $1 \leq i \leq n$.

If a path element contains a dot (.), we can present data about an identification in the database in the following form:

$$i : \{o_1.r_1, \dots, o_n.r_n\}$$

where i is the name of the identification, $o_1.r_1, \dots, o_n.r_n$ are targets of the identification.

The path element $/A:\{B.C, D.E\}$ matches the above identification if A is the same as i , $\{B.C, D.E\}$ is a subset of $\{o_1.r_1, \dots, o_n.r_n\}$. The path element $//\$X:B.C$ matches the above identification if $B.C$ belongs to $\{o_1.r_1, \dots, o_n.r_n\}$ and variable $\$X$ binds to i . The path element $/A:\$X.C$ matches the above identification if A is the same as i and C is the same as r_k with $1 \leq k \leq n$, and variable $\$X$ binds to o_k . The path element $//A:\$X.\Y matches the above identification if A is the same as i , and variable $\$X$ binds to o_k and variable $\$Y$ binds to r_k with $1 \leq k \leq n$. The path element $/\$X:\$Y.\$Z$ matches the above identification with no condition, variable $\$X$ binds to i , variable $\$Y$ binds to o_k and variable

$\$Z$ binds to r_k with $1 \leq k \leq n$.

If a path element does not contain a wildcard (*) or a dot (.), we can present data about either an attribute, a relationship, or an identification in the database in the following form:

$$r : \{o_1, \dots, o_n\}$$

where r is either an attribute name, a relationship name, or an identification name, $o_1, \dots, o_n$ are either values of attributes, targets of relationships or targets of the identification.

The path element $/A:\{B,C\}$ matches the above form if A is the same as r , and $\{B, C\}$ is a subset of $\{o_1, \dots, o_n\}$. The path element $//A:\$X$ matches the above form if A is the same as r , and variable $\$X$ binds o_i with $1 \leq i \leq n$. The path element $\$X:\{B, C\}$ matches the above role relationship if $\{B, C\}$ is a subset of $\{o_1, \dots, o_n\}$, and variable $\$X$ binds to r . The path element $/\$X:\Y matches the above role relationship with no condition, variable $\$X$ binds to r , and variable $\$Y$ binds o_i with $1 \leq i \leq n$.

Path separators in path elements are used to specify searching scope. If the path separator of a path element is a single slash (/), the query manager first explores all attributes and their value types or values stored in the starting node to match the path element. If it successfully matches the path element with an attribute and binds variables to values, then it stops exploring; otherwise, it searches through all of the regular relationships, context relationships, and identifications in the starting node, and explores the corresponding target nodes, context relationship target nodes and identification target nodes. If it successfully matches the path element with one of them and binds variables to values, then it stops exploring. Otherwise, it searches all of role relationship nodes that are children of the starting node and target nodes of role relationships. If it successfully matches the path element with one of the role relationships and binds variables to values, then it stop exploring; otherwise, it fails to match anything in the searching scope and raises an error. If the path separator of a path element is a double slash (//), the query manager traverses the tree whose root is the starting node and does the same procedure, as the path separator is a single slash for each node.

5.6 Query Optimization

Having studied query processing and optimization for XML query languages, we designed our evaluation strategies based on the nature of INM-QL. The query optimization for schema queries is very similar to instance queries. For simplicity, we mainly focus on the instance queries.

Before discussing evaluation strategies, we divide the query expression in instance queries into five groups based on their nature. Each query in the first group has only one literal that contains no string, but variables. Example 27 (1) is a typical case in the first group. Each query in the second group has only one literal that contains one or more class names, a variable and an optional path expression. The path expression must contains no string, but variables. Example 27 (2) is a typical case in the second group, which contains a class name *VicePresident-HumanResources*. Each query in the third group has only one literal that contains an object name in front of a path expression. Example 27 (3) is a typical case in the third group, which has an object name *OH* in front of the path expression. Each query in the forth group has only one literal that contains a variable instead of an object name in front of a path expression that contains some strings. Example 27 (4), Example 27 (5) and Example 27 (6) are typical cases in the forth group, which have variables in front of the path expressions. Each query in the fifth group has two or more literals. Example 27 (7) is a typical case in the fifth group, which has two literals.

Example 27. The following are typical examples of query expressions in instance queries:

- (1) **query** \$X \$Y/\$Z:\$W
- (2) **query** VicePresident-HumanResources \$X/\$Y:\$Z
- (3) **query** Hospital \$X=OH[/VicePresident:*\$Y/age:\$Z | rank:10]
- (4) **query** \$X/worksIn:\$Y/status:Internist
- (5) **query** \$X//VicePresident-HumanResources:Bob/age:45
- (6) **query** Hospital \$X[/Doctor:*\$Y/worksIn:OH, rank:10 | VicePresident:*\$Z]

(7) **query** Hospital \$X=OH//VicePresident:*\$Y, \$Z/age:45

Given a query expression in the query tree structure, the query manager automatically finds out which kind of query expression is and chooses one of following evaluation strategies to process query expression without the user's intervention:

- Sequential search strategy
- Class-object search strategy
- Forward chaining search strategy
- Backward chaining search strategy
- Hybrid search strategy
- Multiple literals search strategy

The sequential search strategy is designed for the first group of query expressions. The class-object search strategy is designated for the second group of query expressions. The forward chaining search strategy is designed for the third group of query expressions. The backward chaining search strategy is designated for the forth group of query expressions that contain only single path. It is always used in the hybrid search strategy, rather than being used independently. The hybrid search strategy is used to process the forth group of query expressions. The multiple literals search strategy is designed to process the fifth group of query expressions. Each strategy may not be applicable to or perform fairly poorly on some cases, but can be particularly efficient for other cases.

5.6.1 Sequential Search Strategy

Query expressions in the first group do not provide any useful information. With sequential search strategy, the query manager sequentially loads objects stored in the object file to the

main memory to process this kind of queries. Sequential search strategy is very expensive when the object file stores a large number of objects.

For example, consider the query in Example 27 (1) and the example data in Figure 5.6. The query consists of three parts: a variable \$X as a class name, a variable \$Y as an object name and a path expression that only contains variables. The query manager sequentially loads objects into the main memory from the beginning of the object file to the end. For each object, it binds variable \$X to classes of the object, variable \$Y to the object, variable \$Z to attribute names, relationship names, identification names, variable \$W to corresponding values, targets and identification targets, respectively.

5.6.2 Class-Object Search Strategy

Query expressions in the second group only provide class names. With class-object search strategy, the query manager first uses class names as keys to search index file of the class file and retrieves IDs of classes. It uses IDs of classes to retrieve sets of IDs of objects from the class-object file, takes the intersection of these sets, and then uses IDs of objects in the intersection to retrieve objects from the object file to process path expressions in these queries.

For example, consider the query in Example 27 (2) and the example data in the Figure 5.3, Figure 5.10, and Figure 5.6. The query consists of three parts: a class name *VicePresident-HumanceResouces*, a variable as an object name and a path expression that only contains variables. The query manager uses class name *VicePresident-HumanceResouces* as a key to search index file of class file and retrieves ID of the class, which is 5. It uses the ID to search class-object file, retrieve an object ID that is 8, and then uses the object ID to retrieve object *Bob* from the object file. The query manager binds variable \$X to *Bob*, and matches the path element */\$Y:\$Z* in the tree structure of object *Bob*. It finally generates a result tree shown in Figure 5.14.

Figure 5.14: A Result Tree Generated for Example 27 (2)

5.6.3 Forward Chaining Search Strategy

With forward chaining search strategy, the query manager first uses the given name of an object in front of the path expression as the key to search the index file of the object file, and then retrieves objects from the database. It uses one root of the obtained object structures as a starting node at a time and follows the path expression in a forward manner to match path elements. After successfully matching a path element and binding variables to values, the query manager has to prepare starting nodes for the next path element in the single path expression. The path element must match either a regular relationship, a role relationship, a context relationship or an identification to provide next starting nodes. The query manager can use IDs in either target nodes, context relationship target nodes, or identification target nodes to retrieve object structures from database as needed.

The two core algorithms of the forward chaining search strategy are shown in Figure 5.15 and Figure 5.16. The first algorithm follows a single path to explore networked objects and to retrieve the results. For simplicity, we only provide a partial algorithm for a pure single path that does not contain a multiple path in the end. The second algorithm essentially does the same as the first algorithm, but it deals with multiple paths. To process a multiple path in a single path, we only need to use the second algorithm in the first algorithm.

Algorithm: ForwardChainingforSinglePath(<i>i</i> , PathTerm, <i>r-tree</i> , <i>r</i>)	
Input:	A node of object tree structure <i>i</i> A PathTerm that contains a double linked list of path elements: <i>p-list</i>
Output:	A result tree: <i>r-tree</i> A boolean <i>r</i> that indicates if search is successful
1	<i>r</i> = FALSE
2	let CurrentElement be the first path element term of <i>p-list</i>
3	let two temporary result trees <i>t-tree</i> =NULL and <i>t-tree2</i> =NULL
4	IF CurrentElement! = NULL and CurrentElement->next == NULL THEN
5	match CurrentElement in <i>i</i> and obtain variable/value bindings
6	IF successfully match CurrentElement THEN
7	<i>r-tree</i> = build a result tree using obtained variable/value bindings
8	<i>r</i> = TRUE
9	ELSE
10	<i>r-tree</i> = NULL and <i>r</i> = FALSE
11	END IF
12	ELSE IF CurrentElement! = NULL and CurrentElement->next! = NULL THEN
13	match CurrentElement in <i>i</i> , obtain variable/value bindings, and prepare a Set of starting nodes for next path element
14	IF successfully match CurrentElement THEN
15	<i>t-tree</i> = build a result tree using obtained variable/value bindings
16	<i>p-list</i> = a list of path elements after CurrentElement
17	FOR each node in Set
18	ForwardChainingforSinglePath(node , <i>p-list</i> , <i>t-tree2</i> , MatchSuccessful)
19	IF MatchSuccessful is TRUE THEN
20	add all subtrees of the empty root of <i>t-tree2</i> as subtrees of a corresponding leaf in <i>t-tree</i> to record valid dependency among variables
21	delete empty root of <i>t-tree2</i> and <i>r</i> = TRUE
22	ELSE
23	recursively delete all nodes that does not match the single path in <i>t-tree</i>
24	END FOR
25	IF <i>r</i> ==TRUE THEN
26	<i>r-tree</i> = <i>t-tree</i>
27	END IF
28	ELSE
29	<i>r-tree</i> = NULL and <i>r</i> = FALSE
30	END IF
31	END IF

Figure 5.15: Algorithm of the Forward Chaining Search Strategy for Single Path

Algorithm: ForwardChainingforMultiplePath(<i>i</i>, MultiPath, <i>r-tree</i>, <i>r</i>)	
Input:	A node of object tree structure <i>i</i> A multiple path term MultiPath
Output:	A result tree: <i>r-tree</i> A boolean r that indicates if search is successful
1	<i>r-tree</i> = NULL and r = FALSE
2	separate single path term in MultiPath into Groups using logic or () as separator
3	FOR each Group in Groups
4	initialize a boolean Match = TRUE, which indicate if matching Group is successful
5	initialize two temporary result trees: <i>t-tree</i> = NULL and <i>t-tree2</i> = NULL
6	FOR each PathTerm in Group
7	ForwardChainingforSinglePath(<i>i</i> , PathTerm , <i>t-tree2</i> , MatchSuccessful)
8	IF MatchSuccessful == FALSE THEN
9	Match =FALSE
10	BREAK
11	ELSE IF MatchSuccessful == TRUE THEN
12	<i>t-tree</i> = merge two result trees <i>t-tree</i> and <i>t-tree2</i>
13	END IF
14	END FOR
15	IF Match ==TRUE THEN
16	<i>r-tree</i> = merge two result trees <i>r-tree</i> and <i>t-tree</i>
17	r =TRUE
18	END IF
19	END FOR

Figure 5.16: Algorithm of the Forward Chaining Search Strategy for Multiple Path

For example, consider the query expression in Example 27 (3) and the example data in Figure 5.6. The query expression consists of four parts: a class name *Hospital*, an object name *OH*, a variable bound to object *OH* and a path expression. First of all, the query manager searches the object file to obtain objects whose names are *OH* and classes are *Hospital*. There is only one hospital named *OH* in the database and its tree structure is shown in Figure 3.5. The value of variable *\$X* is the object *OH*. The path expression contains two single paths with a logical or “|” between them. So we separate these two single paths into two groups. For the first path, there are two path element terms: *//VicePresident:*\$Y* and */age:\$Z*. For the first path element in the first path expression, the query manager traverses the tree structure of the object *OH* to match a role relationship *VicePresident* and obtain its two target nodes *Bob* and *Ben* from its role sub-relationships. The query manager uses IDs in those two target nodes and retrieves objects *Bob* and *Ben* from object file. The values of variable *\$Y* are objects *Bob* and *Ben*, whose tree structures are shown in Figure 3.8. For the second path element, the query manager separately traverses tree structures of object *Bob* and object *Ben* to find attributes named *age* and respectively binds variable *\$Z* to 45 and 55. The value of variable *\$Z* is a value of 45 when *\$Y* binds to *Bob*; the value of variable *\$Z* is equal to 55 when *\$Y* binds to *Ben*. The second path expression is a filtering condition, since it does not have any variable, and it is satisfied by the object *OH* that has an attribute named *rank* with the value of 10. The path expression is therefore evaluated to be logically true, because the query manager successfully matches two groups of single paths. The result tree generated after processing the above query expression is shown in Figure 5.17.

5.6.4 Backward Chaining Search Strategy

The backward chaining search strategy begins with a path element that is at the end of a single path and uses index of object-attribute-relationship file to explore the single path in reverse. The forward chaining search strategy can perform fairly poorly on queries in the

Figure 5.17: A Result Tree Generated for Example 27 (3)

forth group, as it must search the entire database to explore every object and use the root of the object structure as the starting node to process these queries. In order to accelerate searching, the query manager must make sure that it can find the index, in terms of path expression.

The query manager needs to generate search keys for path elements in the single path to search index of object-attribute-relationship file and find corresponding object-attribute-relationship records. For simple cases such as `/attribute:value`, `//relationship:target`, and `//relationship:*target`, the query manager generates three strings: *attribute:value*, *relationship:target* and *:target* as search keys respectively. For normal cases such as `/attribute:variable`, `//relationship:*variable`, `/variable:value`, and `/variable:*target`, the query manager generates four strings: *attribute*, *relationship*, *:value* and *:target* as search keys respectively. When performing backward chaining search, the query manager also combines relationship names in the path elements and object names obtained using the index, in order to generate search keys. Suppose the name of an object is *inst*. If a path element term is a simple case like `//relationship:target`, the search key is *relationship:inst*, where *target* and *inst* are the same name of an object. If a path element term is a simple case like `//relationship:*target`, the search key is *:inst*, where *target* and *inst* are

the same name of an object. If a path element term is a normal case like *//relationship:variable*, the search key is *relationship:inst*. If a path element term is a normal case like *//relationship:*variable*, the search key is *:inst*. If a path element term is a general case like *//variable:variable*, the search key is *:inst*.

With backward chaining search strategy, the query manager first generates a search key for the last path element in the single path, uses the search key to search the index of object-attribute-relationship file, and retrieves object-attribute-relationship records from the database. The instance path in one of object-attribute-relationship records shows name and ID of the object *S* that has an attribute, a relationship or an identification matched the last path element. The query manager then combines the relationship name in the previous path element and the name of *S* to generate the search key and perform the same procedure to explore path elements in reverse. The algorithm of backward chaining search strategy is described in Figure 5.18 and Figure 5.19.

For example, consider the query expression in Example 27 (4) and the example data in Figure 5.9 and Figure 5.6. The query expression consists of two parts: a variable and a path expression. The path expression consists of two path elements: */worksIn:\$Y* and */status:Internist*. The query manager first generates a search key for the last path element */status:Internist*, which is *status:Internist*, searches the index of object-attribute-relationship file, and obtains an object-attribute-relationship record that contains an instance path *((Jack, 11), (OH, 1))*. Because the path separator in */status:Internist* is a single slash (*/*), the query manager uses the ID *1* to retrieve object *OH* and binds variable *\$Y* to *OH*. It then uses the context relationship name in the path element */worksIn:\$Y* and name of the object bound to *\$Y* to generate a search key that is *worksIn:OH*. It uses the search key and obtains two object-attribute-relationship records. These two records contains two instance paths: *((Jack, 11))* and *((Jay, 12))*. It uses two IDs to retrieve objects *Jack* and *Jay*, and binds variable *\$X* to them. The result tree generated after processing the above query expression is shown in Figure 5.20. It must store two variable/value bindings for variable *\$Y* to preserve

<p>Algorithm: BackwardChaining(<i>p-list</i>, <i>V</i>, <i>r-tree</i>, <i>r</i>)</p> <p>Input: A double linked list of path elements: <i>p-list</i> Variable in front of path expression: <i>V</i></p> <p>Output: A result tree: <i>r-tree</i> A boolean <i>r</i> that indicates if search is successful</p> <pre> 1 <i>r-tree</i>=NULL and <i>r</i>=FALSE 2 CurrentElement = last path element term in <i>p-list</i> 3 generate a search Key for CurrentElement 4 use Key and get a set of object-attribute-relationship records from database: Set 5 IF Set is not empty THEN 6 delete CurrentElement from <i>p-list</i> 7 FOR each Record in Set 8 initialize a temporary result tree: <i>t-tree</i> = NULL 9 InstancePath = instance path of Record 10 BackwardChainingHelper(<i>p-list</i>, <i>V</i>, InstancePath, NULL, <i>t-tree</i>, MatchSuccessful) 11 IF MatchSuccessful == TRUE THEN 12 <i>r-tree</i> = merge two result trees <i>r-tree</i> and <i>t-tree</i> 13 <i>r</i> = TRUE 14 END IF 15 END FOR 16 END IF </pre>
--

Figure 5.18: Algorithm of the Backward Chaining Search Strategy

<p>Algorithm: BackwardChainingHelper(<i>p-list</i>, <i>V</i>, <i>Path</i>, <i>t-tree</i>, <i>r-tree</i>, <i>r</i>)</p> <p>Input: A double linked list of path elements: <i>p-list</i> Variable in front of path expression: <i>V</i> Instance Path of object-attribute-relationship record: <i>Path</i> A temporary result tree: <i>t-tree</i></p> <p>Output: A result tree:<i>r-tree</i> A boolean <i>r</i> that indicates if search is successful</p> <pre> 1 <i>r-tree</i> = NULL and <i>r</i> = FALSE 2 IF <i>p-list</i> == NULL THEN 3 get the object from database using the first object ID in <i>Path</i> 4 bind <i>V</i> to object 5 store the variable/value binding in the root of <i>t-tree</i> 6 <i>t-tree</i> = add a empty root for <i>t-tree</i> 7 <i>r-tree</i> = <i>t-tree</i> and <i>r</i> = TRUE 8 ELSE 9 CurrentElement = last path element term in <i>p-list</i> 10 get the object from database using an ID in <i>Path</i> in terms of the path separator in CurrentElement 11 IF target of CurrentElement is a variable THEN 12 bind variable to object 13 store the variable/value binding in the root of <i>t-tree</i> 14 <i>t-tree</i> = add a empty root for <i>t-tree</i> 15 ELSE IF name of object does not match target of CurrentElement THEN 16 <i>r-tree</i> = NULL and <i>r</i> = FALSE 17 RETURN 18 END IF 19 use the relationship name of CurrentElement and the name of object as target to generate a search key: Key 20 use Key and get a set of object-attribute-relationship records from database: Set 21 IF Set is empty THEN 22 <i>r-tree</i> = NULL and <i>r</i> = FALSE 23 ELSE 24 initialize another temporary result tree: <i>t-tree2</i> = NULL 25 delete CurrentElement from <i>p-list</i> 26 FOR each Record in Set 27 InstancePath=instance path of Record 28 BackwardChainingHelper(<i>p-list</i>, <i>V</i>, InstancePath, <i>t-tree</i>, <i>t-tree2</i>, MatchSuccessful) 29 IF MatchSuccessful == TRUE THEN 30 <i>r-tree</i> = merge two result trees <i>r-tree</i> and <i>t-tree2</i>) 31 <i>r</i>=TRUE 32 END IF 33 END FOR 34 END IF 35 END IF </pre>

Figure 5.19: Algorithm of Backward Chaining Helper

Figure 5.20: A Result Tree Generated for Example 27 (4)

correct dependency among variables.

The advantage of the backward chaining search strategy is that it always starts with objects that have attributes, relationships or identifications matched the last path elements in the single path and avoids needlessly exploring unsatisfying objects. However, the backward chaining search strategy can perform fairly poorly when there could be many objects satisfying the last path elements, but very few of those objects match the path expression.

5.6.5 Hybrid Search Strategy

To process a single path term with the hybrid search strategy, the query manager selects a path element as the starting element in the single path term and generates a search key for the starting element. It uses the search key to search the index of the object-attribute-relationship file and retrieve object-attribute-relationship records from database. For each object-attribute-relationship record, it contains an instance path that shows the name and ID of the object S that has a satisfying relationship and a target T . It contains the ID of the target T . If the object-attribute-relationship record is a record of a regular relationship or a role relationship, the query manager retrieves the target using ID, uses the root of the target as the starting node and explores part of the single path expression after the starting element, with the forward chaining search strategy. If the object-attribute-relationship record is a record of a context relationship or an identification, the query manager obtains the target

node of the context relationship or the identification in the object S , and also retrieves the target with ID . It uses the target node and the root of the target as the starting nodes and explores the single path expression after the starting element with the forward chaining search strategy. The query manager then explores path elements before the starting element with the backward chaining search strategy.

There could be several simple cases, normal cases, and general cases in one single path expression. It is not reasonable to randomly choose a simple case, a normal case or a general case as the starting element and start query processing with the hybrid search strategy. Simple cases contains more information than normal cases and normal cases contain more information than general cases. But it does not necessarily mean that we should always choose simple cases as the starting element. It is highly possible that there exists one normal case, and the number of object-attribute-relationship records that correspond to it is much less than the number of records for other simple cases. In this case, to choose that normal case as the starting element could be superior to choosing other simple cases. However, the number of object-attribute-relationship records that correspond to a simple case or a normal case is not the only factor when choosing the start element. The position of a path element term is another important factor. Consider that there could be one simple case in a path expression; it relates the least number of object-attribute-relationship records, but it locates at the end of the path expression. If we choose this simple case as the starting element, it leads to several times of searching the index and backward chaining, which could be very costly. In this case, it is better to find a simple case or a normal case, which relates to not too many of the object-attribute-relationship records and has a position near to the front of query expression. Therefore, we introduce a function to evaluate every path element in order to choose a starting element in the a path expression.

The evaluation function balances all factors that matter for choosing a path element to

be a starting element. The following is the evaluation function:

$$\mathcal{F} = \frac{\mathcal{W}}{\mathcal{D}^{\mathcal{K}} \times \mathcal{N}} \quad (5.1)$$

where:

- $\mathcal{W}$ is pre-assigned weight of a path element. Note that $\mathcal{W}_S > \mathcal{W}_N > \mathcal{W}_G$, where $\mathcal{W}_S$ denotes weight of simple case, $\mathcal{W}_N$ denotes weight of normal case, and $\mathcal{W}_G$ denotes weight of general case.;
- $\mathcal{D}$ is the depth of path element term in a path expression;
- $\mathcal{K}$ is a coefficient to adjust the impact of $\mathcal{D}$ on the result of evaluation and $\mathcal{K} > 1$;
- $\mathcal{N}$ is the number of object-attribute-relationship records, which correspond to the path element.

Two core algorithms of the hybrid search strategy of are shown in Figure 5.21 and Figure 5.22. The first algorithm deals mainly with single path. The second algorithm uses the first algorithm to deal with multiple path.

For example, consider the query expression in Example 27 (5) and the example data in Figure 5.9 and Figure 5.6. The query expression consists of two parts: a variable and a path expression. The path expression consists of two path elements: */VicePresident-HumanResources:Bob* and */age:45*. The query manager first generates search keys for path elements and uses the evaluation function to evaluate them. For the first path element */VicePresident-HumanResources:Bob* whose position is 1, the query manager generates the search key *VicePresident-HumanResources:Bob*, and uses the key to obtain one object-attribute-relationship record. So the final score of the second path element is $\frac{\mathcal{W}_S}{1^{\mathcal{K}} \times 1}$. For the second path element */age:45* whose position is 2, its final score after evaluation is $\frac{\mathcal{W}_S}{2^{\mathcal{K}} \times 1}$. The first path element has the highest score and is chosen as the starting element in the single path. The object-attribute-relationship record that relates to it is in Figure 5.9.

Algorithm: HybridSearchforSinglePath(PathTerm, V, r-tree, r)	
Input:	A path term PathTerm that contains a double linked list path element terms: <i>p-list</i> Variable in front of path expression: V
Output:	A result tree: <i>r-tree</i> A boolean r that indicates if search is successful
1	<i>r-tree</i> =NULL and r =FALSE
2	use evaluate function to evaluate every path element in <i>p-list</i>
3	StartingElement = the path element with highest score
4	<i>p-list2</i> = a double linked list of path element terms before StartingElement
5	<i>p-list3</i> = a double linked list of path element terms after StartingElement
6	Key = generate a search key for StartingElement
7	Set = get a set of object-attribute-relationship records with Key
8	FOR each Record in Set
9	InstancePath = instance path of Record
10	initialize two temporary result trees <i>t-tree</i> = NULL and <i>t-tree2</i> = NULL
11	IF <i>p-list3</i> !=NULL THEN
12	Targets = prepare starting nodes in terms of relationships stored in Record
13	FOR each Target in Targets
14	ForwardChainingforSinglePath(Target , <i>p-list3</i> , <i>t-tree</i> , MatchSuccessful)
15	IF MatchSuccessful == TRUE THEN
16	BackwardChainingHelper(<i>p-list2</i> , V , InstancePath , <i>t-tree</i> , <i>t-tree2</i> , MatchSuccessful)
17	IF MatchSuccessful == TRUE THEN
18	r =TRUE
19	<i>r-tree</i> = merge two result trees <i>r-tree</i> and <i>t-tree2</i>
20	END IF
21	END IF
22	END FOR
23	ELSE IF <i>p-list3</i> ==NULL THEN
24	BackwardChainingHelper(<i>p-list2</i> , V , InstancePath , <i>t-tree</i> , <i>t-tree2</i> , MatchSuccessful)
25	IF MatchSuccessful == TRUE THEN
26	r =TRUE
27	<i>r-tree</i> = merge two result trees <i>r-tree</i> and <i>t-tree</i>
28	END IF
29	END IF
30	END FOR

Figure 5.21: Algorithm of the Hybrid Search Strategy for Single Path

In the record, the target value of relationship is $(Bob, 8)$, and the instance path is $((OH, 1), (VicePresident, 19))$, where the ID of object Bob is 8 and the ID of object OH is 1. The query manager use two IDs to retrieve objects into main memory. It uses the root of the tree structure of object Bob as the starting node, explores path expression after the starting element and successfully matches path element $/age:45$ in object Bob . It finds the starting element is the first element in the single path and finally binds variable $\$X$ to object OH .

For example, consider the query expression in Example 27 (6) and example data in Figure 5.6. This query expression contains multiple paths. The query manager separates single path terms into groups by using logical or “|” as a separator. The first group consists of two single path terms: $//Doctor:*\$Y/worksIn:OH$ and $rank:10$. The evaluation function is used to evaluate every path element of single path terms in the first group. There is only one path element in the second single path term in the first group. We generate the search key $rank:10$ for path element $rank:10$, and the query manager obtains one record in object-attribute-relationship file shown in Figure 5.9. The position of path element $rank:10$ is 1, hence its final score after evaluation is $\frac{w_s}{1 \times x_1}$. The final scores of two path elements of the first single path term in the first group are $\frac{w_N}{1 \times x_1}$ and $\frac{w_s}{2 \times x_2}$. The path element $rank:10$ has the highest score and is chosen as the starting element in the first group. The object-attribute-relationship record that relates to it is in Figure 5.9. The instance path in the record is $((OH, 1))$, and ID of object OH is 1 that is used to retrieve it from the database. The variable $\$X$ binds to object OH . The the root of object OH is used as a starting node for sequentially matching and searching the rest of the single path terms in the first group. $Doctor$ is found as a role relationship and its role sub-relationships have targets $Jack$ and Jay . The query manager uses IDs stored in the target nodes of role relationships to retrieve objects $Jack$ and Jay , and binds variable $\$Y$ to them. Both $Jack$ and Jay work in OH . A temporary result tree is build for the first group of single path terms and it is shown in Figure 5.23(a). For the second group, the query manager generates a search key for $VicePresident:*\$Z$, which is $VicePresident$, and obtains one object-attribute-relationship record that contains

<p>Algorithm: HybridSearchforMultiplePath(MultiPath, V, r-tree, r)</p> <p>Input: Multiple path Term: MultiPath Variable in front of path expression: V</p> <p>Output: A result tree: <i>r-tree</i> A boolean r that indicates if search is successful</p> <pre> 1 <i>r-tree</i>=NULL and r=FALSE 2 separate single path term in MultiPath into Groups using logic or as separator 3 FOR each Group in Groups 4 initialize a boolean Match=FALSE 5 initialize two temporary result trees:<i>t-tree</i>=NULL and <i>t-tree2</i>=NULL 6 FOR each PathTerm in Group 7 use evaluation function to evaluate every path element in PathTerm 8 END FOR 9 PathTerm = the single path term that has the path element with highest score in Group 10 HybridSearchforSinglePath(PathTerm, V, <i>t-tree</i>, MatchSuccessful) 11 IF MatchSuccessful == FALSE THEN 12 BREAK 13 ELSE 14 delete PathTerm from Group 15 Set = the roots of all objects bound to V in <i>t-tree</i> 16 FOR each object in Set 17 initialize a boolean MatchObject=TRUE 18 FOR each PathTerm in Group 19 ForwardChainingforSinglePath(object, PathTerm, <i>t-tree2</i>, MatchSuccessful) 20 IF MatchSuccessful == FALSE THEN 21 delete node that contains object and all its children in <i>t-tree</i> 22 MatchObject = FALSE 23 BREAK 24 ELSE 25 <i>t-tree</i> = merge two result trees <i>t-tree</i> and <i>t-tree2</i> 26 END IF 27 END FOR 28 IF MatchObject==TRUE THEN 29 Match=TURE 30 END IF 31 END FOR 32 END IF 33 IF Match==TRUE THEN 34 <i>r-tree</i> = merge two result trees <i>r-tree</i> and <i>t-tree</i> 35 r=TRUE 36 END IF 37 END FOR </pre>

Figure 5.22: Algorithm of the Hybrid Search Strategy for Multiple Path

Figure 5.23: Result trees generated with the hybrid search strategy

an instance path $((OH, 1))$. It uses the ID to retrieve the object OH . The object OH has a role relationship named *VicePresident*, whose role sub-relationships have targets Bob and Ben . It uses IDs stored in the target nodes of role relationships to retrieve objects Bob and Ben , and binds the variable $\$Z$ to them. Another temporary result tree is generated, which is shown in Figure 5.23(b). Finally, two temporary result trees are merged into one result tree that is shown in Figure 5.23(c).

5.6.6 Multiple Literals Search Strategy

The multiple literals search strategy is designed to process some queries that have two or more literals. None of the above strategies alone is good for these queries. The sequential search strategy, the class-object search strategy, the forward chaining search strategy and the hybrid search strategy are suitable for different literals. Each strategy mentioned above is particularly efficient for processing some literals, but may not be applicable to or perform relatively poorly on others. The query manager effectively combines above strategies together in the multiple literals search strategy. It can automatically alternate mechanisms to process literals without any user's intervention. Moreover, it is able to generate all answers and respond the user at reasonable speed. The brief algorithm of the multiple literals search strategy is described in Figure 5.24.

Algorithm: MultipleLiteralsSearch(<i>l-list</i>, <i>r-tree</i>, <i>r</i>)	
Input:	a list of literals: <i>l-list</i>
Output:	A result tree: <i>r-tree</i>
	A boolean <i>r</i> that indicates if search is successful
1	<i>r-tree</i> = NULL and <i>r</i> = TRUE
2	Term = the first literal in <i>l-list</i>
3	WHILE Term ->next!=NULL
4	a temporary result tree <i>t-tree</i> = NULL
5	IF Term provides name of an object in front of path expression THEN
6	Objects = get objects from database using name
7	choose forward chaining search strategy and use the roots of Objects
	as starting point nodes to match Term
8	IF forward chaining search is successful THEN
9	<i>t-tree</i> = returned result tree of the forward chaining search strategy
10	<i>r-tree</i> = merge two result trees <i>r-tree</i> and <i>t-tree</i>
11	ELSE
12	<i>r-tree</i> = NULL and <i>r</i> = FALSE
13	BREAK
14	END IF
15	ELSE IF Term provides variable in front of path expression THEN
16	IF there exists values of variable in <i>r-tree</i> THEN
17	Objects = get objects bound to variable in <i>r-tree</i>
18	choose forward chaining search strategy and use the roots of
	Objects as starting nodes to match Term
19	IF forward chaining search is successful THEN
20	<i>t-tree</i> = returned result tree of the forward chaining search strategy
21	<i>r-tree</i> = merge two result trees <i>r-tree</i> and <i>t-tree</i>
22	ELSE
23	<i>r-tree</i> = NULL and <i>r</i> = FALSE
24	BREAK
25	END IF
26	ELSE
27	choose either sequential search strategy, class-object search strategy,
	or the hybrid search strategy to match Term according to the nature of Term
28	IF search is successful THEN
29	<i>t-tree</i> = returned result tree of one search strategy
30	<i>r-tree</i> = merge two result trees <i>r-tree</i> and <i>t-tree</i>
31	ELSE
32	<i>r-tree</i> = NULL and <i>r</i> = FALSE
33	BREAK
34	END IF
35	END IF
36	END IF
37	Term = Term ->next
38	END WHILE

Figure 5.24: Algorithm of the Multiple Literals Search Strategy

For example, consider Example 27 (7) and example data in Figure 5.6. The query manager chooses the forward chaining strategy to process the first literal:

Hospital \$X=OH//VicePresident:*\$Y.

It first finds one object *OH* that belongs to class *Hospital* from database, and binds variable \$X to object *OH*. It then uses the root of object *OH* as a starting node to explore the path expression in a forward manner, finds a role relationship *VicePresident* in *OH*, retrieves the targets of *VicePresident* and role sub-relationship of *VicePresident* with IDs stored in the corresponding target nodes, and binds variable \$Y to them. It then generates first temporary result tree that is shown in Figure 5.25(a). For the second literal \$Z/age:45, the query manager does not find the values of variable \$Z in the first temporary result tree and chooses the hybrid search strategy to process the second term. Since there is only one path element in the second literal, the query manager first generates a search key *age:45* for path element /age:45 and finds one record in object-attribute-relationship file shown in Figure 5.9. The record shows object *Bob* has an attribute *age* with the value of 45. It retrieve object *Bob* with ID, binds variable \$Z to object *Bob*, and generates second result tree that is shown in Figure 5.25(b). Finally, it merges two temporary result trees into a final result tree that is shown in Figure 5.25(c).

As literal processing depends on the its nature, the sequential search strategy, the class-object search strategy, the forward chaining search strategy and hybrid search strategy are alternated a number of times to answer the query.

5.7 Query Output Handler

Query output handler uses result tree to generate query results in terms of the return pattern. It supports many operations such as aggregate, order by and grouping, and many kinds of construction terms to form query results. Construction terms can be used nested in other construction terms; different algorithms are used to deal with different construction terms.

Figure 5.25: Result Trees Generated by the Multiple Literals Search Strategy

An aggregation term is used to compute a single value from a collection of values bound to a variable. We can use the name of the variable and quickly find all of its variable/value bindings in the result tree with the hash table of variable/values. The query output handler checks if the values are numeric for aggregate functions *avg*, *sum*, *min* and *max*, and it raises an error if they are not.

Query 11. The query manager generates a result tree shown in Figure 5.17 for the following query:

query Hospital \$X=OH[/"Vice-President":*\$Y/age:\$Z | rank:10]

construct "Average age of vice-presidents:"avg(\$Z)

Result: Average age of vice-presidents:50

For Query 11, the query output handler uses \$Z as the key to look for nodes that contain values of variable \$Z in the hash table of variable/values, retrieves values from nodes: 45 and 55, and calculates average of these two values, which is 50. It finally displays the description string and calculated result.

As we described before, a general term has the form $d_1 x d_2$ order by $T_1 t_1, \dots, T_n t_n$. It sorts values bound to variable x by values of $T_1, \dots, T_n$. With the hash table of variable/values, the query output handler uses variables in $T_1, / \dots /, T_n$ to find nodes in the

result tree for them. Nodes used by $T_1, \dots, T_n$ must be descendants of nodes used by variable x . If this is not true, the query output handler raises an error. It is also necessary to consider the type of values of T_i with $1 \leq i \leq n$ when sorting them. All the values sorted must have comparable types. The query output handler automatically checks data type for sorting values. For example, values could be all integers or all strings. It is valid if values are a mix of integers and decimals, since values of these types can be compared. However, a type error is raised if values are a mix of integers and strings.

For example, the query manager generates a result tree for Query 9 in Chapter 4, which is the same as the result tree of Query 11. The query output handler first uses $\$Y$ as a key to look for nodes in the hash table of variable/values and retrieves values from nodes: *Bob* and *Ben*. It then finds values of variable $\$Z$ in subnodes of *Bob* and *Ben*, which are 45 and 55 respectively. It finally changes order of *Bob* and *Ben* and displays.

A path term is used to display values of variables and dependency among them in a path form. As we described, it has the form $T_1 / \dots / T_n$. With the hash table of variable/values, the query output handler uses variables in $T_1, \dots, T_n$ to find nodes in the result tree for them. Nodes used by T_i must be ancestors of nodes used by T_{i+1} with $1 \leq i \leq n$. If this is not true, the query output handler raises an error. In a path, all $T_1, \dots, T_n$ must be single-valued. If two or more values of T_{i+1} are dependent on one value of T_i with $1 \leq i \leq n$, the query output handler displays several paths that show all combinations.

Query 12. The query manager generates a result tree shown in Figure 5.23(c) for the following query:

```

query Hospital $X[/Doctor:*$Y/worksIn:OH, rank:10 | VicePresident:*$Z]
construct $X/Doctor:$Y
Result: OH
           Doctor:Jack,
OH
           Doctor:Jay

```

For Query 12, the query output handler uses the key $\$X$, finds a node that contains a value of variable $\$X$ in hash table of variable/values and retrieves the value from the node

OH. It then uses the key *\$Y* and finds two nodes that contain two values of variable *\$Y*, confirms these two nodes are descendants of the node that contains *\$X/OH* and retrieves values *Jack* and *Jay*. As two values of variable *\$Y* are dependent on a value of variable *\$X* in a path term, the query output handler finally displays two pathes that show two combinations.

A grouping term is used to build several values into a set. It is normally used in other construction term.

Query 13. The following query has the same query expression as in Query 12, so it has the same result tree as Query 12. But it has a different construction term that contains a grouping term.

query Hospital *\$X*[//Doctor:*\$Y/worksIn:OH, rank:10 | VicePresident:*\$Z]

construct *\$X*/Doctors:{*\$Y*}

Result: OH

Doctors:

Jack

Jay

For Query 13, the query output handler acts almost the same as in Query 8. But it groups *Jack* and *Jay* into a set that is dependent on *OH*. Hence the query output handler display one path instead of two paths in Query 12.

A tuple term provides a way to combine several path terms that start with the same variable and it displays values of variables and dependency among them in a tuple form. As we described, it has the form $T [T_1, \dots, T_n]$. With the hash table of variable/values, the query output handler uses variables in $T, T_1, \dots, T_n$ to find nodes in the result tree for them. Nodes used by T must be ancestors of nodes used by $T_1, \dots, T_n$. If this not true, the query output handler raises an error.

Query 14. The following query has the same query expression as in Query 12, so it has the same result tree as Query 12. But it has a tuple term in result construction expression.

query Hospital *\$X*[//Doctor:*\$Y/worksIn:OH, rank:10 | VicePresident:*\$Z]

construct *\$X*[Doctors:{*\$Y*}, "Number of Vice-Presidents:"count({*\$Z*})]

Result: OH[

Doctors:

```

      Jack
      Jay,
Number of Vice-Presidents:2]

```

For Query 14, the query output handler acts the same as in Query 9 for general terms $\$X$ and $\text{Doctors:}\{\$Y\}$. It then uses the key $\$Z$, finds nodes that contain two values of variable $\$Z$, confirms that these nodes are descendants of the node that contains $\$X/OH$ and obtains two nodes. It finally combines all found information and descriptions in to a tuple form.

A list term provides a way to display values of variables without any dependency among them. As we described, it has the form $T_1, \dots, T_n$. The query output handler does not check dependency among $T_1, \dots, T_n$.

Query 15. The following query has the same query expression as in Query 8, so it has the same result tree as Query 8. But it has a list term in result construction expression.

```

query Hospital  $\$X[//\text{Doctor}:\ast\$Y/\text{worksIn}:\text{OH}, \text{rank}:10 \mid \text{VicePresident}:\ast\$Z]$ 
construct  $\$X[\text{Doctors:}\{\$Y\}, \text{Vice-Presidents:}(\{\$Z\})], \$Z[]$ 
Result: OH[

```

```

    Doctors:
      Jack
      Jay,
    Number of Vice-Presidents:2],
  Bob[age:45, gender:male,
    position:OH.VicePresident-HumanResources[startYear:2007]],
  Ben[age:55, gender:male, health:OH.Patient[P#:002],
    position:OH.VicePresident-HumanResources[startYear:2008]]

```

For Query 15, the query output handler acts the same for the tuple term $\$X[\text{Doctors:}\{\$Y\}, \text{Vice-Presidents:}(\{\$Z\})]$. It then uses the key $\$Z$, finds nodes that contain two values of variable $\$Z$, confirms that these nodes are descendants of the node that contains $\$X/OH$ and obtains values that are *Bob* and *Ben*. It then retrieves objects *Bob* and *Ben* from the database to display all of the information of two objects. It finally combines all found information and descriptions in to a list form.

A pair term is used to display either names and values of attributes, names and targets of relationships, and names and targets of identifications. It has the form $x : y$. With the

Figure 5.26: A Result Tree of Query 12

hash table of variable/values, the query output handler uses variables in X and y to find nodes in the result tree for them. Nodes used by x must be ancestors of nodes used by y . If this not true, the query output handler raise an error.

Query 16. The query manager generates a result tree shown in Figure 5.26 for the following query:

query OH/VicePresident:*\$X/\$Y:\$Z

construct VicePresident:\$X/{ \$Y:\$Z }

Result: VicePresident:

Bob

age:45

gender:male

position:OH.VicePresident-HumanResources

VicePresident:

Ben

age:55

gender:male

position:OH.VicePresident-MedicalAffairs

health:OH.Patient

For Query 16, the query output handler uses the key $\$X$, finds nodes that contain two values of variable $\$X$ in hash table of variable/values, and obtains values that are *Bob* and *Ben*. It then uses the key $\$Y$ and finds six nodes that contain six values of variable $\$Y$. It confirms that three nodes of variable $\$Y$ are descendants of the node that contains $\$X/Bob$ and that the other three nodes of variable $\$Y$ are descendants of node that contains $\$X/Ben$.

It then do the same procedure to confirm that nodes of variable \$Z have corresponding nodes of variable \$Y. The query output handler builds pairs using nodes of variables \$Y and \$Z, and it finally groups pairs in a path term.

5.8 User Interface

We develop the client that has graphical user interfaces for the user to connect to the database running on the server and retrieve information stored in the INM database. We take an example to illustrate how to use such graphical user interface of the client. Before discussing examples, we need to use INM-DDL and INM-DML to model hospital information illustrated in Figure 3.1 and Figure 3.4.

We provide a browse console to browse all kinds of modeling information and automatically generated information in almost all files. The browse console for class file lists all of classes including object classes and reduced role relationship classes in the class file, as shown in Figure 5.27. The browser console for the class file is also able to provide more detail for individual classes by double clicking on class names. The Figure 5.28 shows all information and structure of class *Hospital*.

The browse console for object file lists all of objects in the object file as shown in Figure 5.29. The browse console for object file is also able to provide more detail for individual objects by double clicking on object names. Figure 5.30 shows all of the information and structure of object *OH*.

We can conduct Query 8 in search console. The Figure 5.31 demonstrates that we can attain two vice-presidents in hospital *OH*, and their average age, maximum age, and minimum age are 50, 55, and 45 respectively.

Figure 5.27: Browser console for class file

Figure 5.28: Browser console for class Hospital

Figure 5.29: Browser console for object file

Figure 5.30: Browse console for object OH

Figure 5.31: INM Query Console

5.9 Experimental Results

Our experimental platform is a debian/linux 2.6 operation system that runs on a 2.8 GHz Intel Pentium dual core PC with 1G of physical memory. We uses INM-DDL to define classes and import objects with INM-DML. Imported objects that conform the schema of hospital information. Objects that belong to class *Hospital* have the same attributes with the same values as hospital *OH* and have the same relationships to relate with seven other objects. These objects have the same attributes with the same values as objects related with hospital *OH*. All imported objects have unique names and are stored in the INM database that is built on the Berkeley DB. Additionally, the parameters we used in the evaluation function are set to be following values: $\mathcal{W}_S = 10000$, $\mathcal{W}_N = 100$, $\mathcal{W}_G = 1$ and $\mathcal{K} = 2$.

Note that following query time shows how long the server takes to process queries and it does not include time of communication between the client and the server.

Number of objects	Size of database
800	2.58MB
4000	12.27MB
8000	24.6MB
16000	45.16MB

Table 5.1: Size of database as number of objects scales

Experiment One Table 5.1 records the number of objects and the database size. Figure 5.32 illustrates the size of the database scales as the number of objects scales. We can observe that size of database linearly increase as the number of objects scales.

Figure 5.32: Storage size as number of objects scales

Experiment Two We conduct the following query:

query Hospital \$X=OH[/VicePresident:*\$Y/age:\$Z | rank:10]

construct \$X/\$Y/\$Z;

The table 5.4 records query times, the number of objects and the number of queried objects. Figure 5.35 illustrates how the query time changes as the number of objects scales. We can observe that query time slightly increases as number of objects scales when the number of queried objects remains the same, as the time of searching object *OH* increases as the object file increases.

Number of objects	Number of Queried Objects	Query Time
800	3	0.738ms
4000	3	0.758ms
8000	3	0.778ms
16000	3	0.79ms

Table 5.2: Query Time and Number of Queried Objects as Number of Objects Scales in Experiment Two

Figure 5.33: Query Time of Experiment Two

Number of objects	Number of Queried Objects	Query Time
800	300	20.994ms
4000	1500	135.546ms
8000	3000	366.195ms
16000	6000	1120.782ms

Table 5.3: Query Time and Number of Queried Objects as Number of Objects Scales in Experiment Three

Experiment Three We conduct the following query:

query \$X/VicePresident:*\$Y

construct \$X/\$Y;

The table 5.4 records query times, the number of objects and the number of queried objects.

Figure 5.35 illustrates how the query time changes as the number of object scales. We can observe that the query speed slows down as the number of queried objects scales, since the query manager spends more and more time on building result tree.

Figure 5.34: Query Time of Experiment Three

Experiment Four We conduct the following query:

query Hospital \$X = OH //VicePresident:*\$Y, \$Z/age:45

construct \$X/\$Y[], \$Z[];

The table 5.4 records query times, the number of objects and the number of queried objects.

Number of objects	Number of Queried Objects	Query Time
800	103	9.43ms
4000	503	76.137ms
8000	1001	209.412ms
16000	2003	1466.797ms

Table 5.4: Query Time and Number of Queried Objects as Number of Objects Scales in Experiment Four

Figure 5.35 illustrates how the query time changes as the number of object scales. We can observe that the query speed slows down as the number of queried objects scales, as the query manager spends more and more time on building the result tree and generating the query results.

Figure 5.35: Query Time of Experiment Four

Chapter 6

Conclusion

INM naturally and directly supports diverse kinds of relationships between objects, between relationships, and between objects and relationships. It is able to represent various roles that objects play via these complex relationships. It also enables us to represent not only static but also dynamic context-dependent information about objects.

INM-QL is specially designed for INM to retrieve information about schema and instance, their attributes, relationships and context-dependent information from the database. It has following features:

- INM-QL consists of schema queries and instance queries. Schema queries are used to explore networked classes at the schema level and to retrieve data of classes, their attribute, relationships, subclasses and context-dependent information. Instance queries are designed to explore networked objects at the instance level and to retrieve data of objects, their attributes, relationships and context-dependent information.
- Variables are all logical variables that are place holders and have no types. They can be used to bind to anything of objects based on its locations in the query, which make INM-QL flexible and easy to use in practice.
- INM-QL supports single and multiple path expressions to explore networked classes and objects at the schema level and at the instance level.

- Attributes and various relationships are treated in the same way. The system automatically figures out which are attributes, which are relationships, and their kinds; hence, the user is not required to explicitly specify attributes and various relationships.
- INM-QL provides result construction expression to process query results in the user specified form. It also supports operations, such as order by, aggregate and grouping functions, which are integrated into result construction expression.

INM database management system has been implemented, which uses the thin client/fat server architecture and supports INM-DDL, INM-DML and INM-QL. The thesis is mainly focused on the implementation of the subsystem that can process INM-QL in INM database management system. The subsystem has following features:

- It has a lexical and syntactical analyzer of INM-QL, which produces a stream of tokens, verify that the token stream is syntactically correct, and then construct a valid parse tree for the entire program.
- It uses an intermediate result structure to hold the intermediate query results and other supporting structures to reduce the complexity of query processing.
- It uses established index in the database to speed up query processing.
- It is able to process INM-QL effectively and efficiently. It automatically decides and chooses one appropriate evaluation strategy to process INM-QL without user's intervention, based on the nature of the query and knowledge of data in the database.
- It supports many construction terms and build-in functions, such as aggregate, order by and grouping. We had designed and implemented different algorithms to process query results in the user specified form, so that the subsystem can answer queries within a reasonable amount of time.

- It is an independent platform that is specially designed for INM and it is very easy to extend.

We can also extend our system in the following ways:

- We could be able to add quantifiers, various predefined functions, conditional expressions and used defined functions into our INM-QL. Those features are very similar to XQuery and can make our INM-QL more powerful and easier for the user to use.
- We can further optimize our query manager. we can find and apply advanced evaluation strategies to fit more particular cases and powerful index to save storage space and speed up query processing.
- We can enable our INM-QL to support the keyword search and to provide more user friendly interfaces. INM-QL can return top-k results, instead of building the entire results and returning at one time.

Appendix A

BNF of INM-QL

Query Language	
Query	::="query"(instanceQuery schemaQuery) [construct]"",
Schema Query	
schemaQuery	::= type (variable[oConTar [",",secSchQuery]] [variable "="] contextName [oConTar [",",secSchQuery]])
Type	::= "class" "tuple"
oConTar	::= ("/" "//") oPathCon oMultiPath [consTar] osubClassTar consTar
oPathCon	::= oPathElement {("/" "//")oPathElement }[(oMultiPath consTar)]
oPathElement	::= ["!"] name variable="contextName ["!"] [name [oMultiPath]] ":"(["!"] contextName variable valueType) variable [oMultiPath] ":" (variable valueType) [attrRelType] variable
valueType	::= "INT" "STRING" ...
oMultiPath	::= "[" ["/"] oPathCon {("," " ") oPathCon } "]"
osubClassTar	::= "isa" ["*"] (variable contextName "{ " contextName " , " contextName { " , " contextName } " } " "(" contextName (" , " " ") contextName { (" , " " ") contextName } ") ")
consTar	::= "(" modifier variable { " , " modifier variable } ") "
modifier	::= "inverse" "identification" "prerequisite" "migrationto" "constraint"
secSchQuery	::= (variable contextName) oConTar [",",secSchQuery]

Table A.1: BNF of Schema Query Expression

Instance Query	
instanceQuery	::=[classNames]([variable=""] contextName variable)[condition ["," secQuery]]
classNames	::= contextName["*"] "("contextName["*"] ("," " ") ["!"]con- textName["*"] {"("," " ") ["!"]contextName["*"]} ")"
contextName	::= name{"." Name}
name	::= String "" String ""
condition	::= ("/" "/") path multiPath
path	::= pathElement {"/" "/"}pathElement}[multiPath]
pathElement	::= ["!"] name [attriRelType] variable ["!"] name [multiPath] ":"["*"] arConValues ["!"] name "IS" "NULL" ["!"] name relop arConValues ["!"] ":"["*"] arConValues ["!"] [name[multiPath]] ":"["*"] ["!"] classNames(arConValues variable) ["!"] [name[multiPath]] ":"["*"] arTarValue variable [multiPath] ":"["*"] arConValues variable [multiPath] ":"["*"] variable
relop	::= "<" "<=" ">" ">=" "<>" "contains"
attriRelType	::= attriType relType
attriType	::= [("normal" "local" "c-based" "specific" "rel" "static")] "@"
relType	::= [("normal" "role" "c-based" "specific" "context" "cdi" "iden"] "#"
arConValues	::= arConValue ARMulConValue ARListConValue
arMulConValue	::= "(" arConValue ("," " ") arConValue {"("," " ") arConValue} ")"
arListConValue	::= "{" arConValue "," arConValue "}"
arConValue	::= [variable "="] contextName
arTarValue	::= {name"."} variable {"." name}
multiPath	::= "[""/"] path ("," " ") ["/"]path {"("," " ") ["/"]path }"]
secQuery	::= secConTar ["," secQuery]
aClassNames	::= ["!"]contextName["*"] "(" ["!"]contextName["*"] ("," " ") ["!"]contextName["*"] {"("," " ") ["!"]contextName["*"]} ")"
secConTar	::= [aClassNames] (variable contextName)[condition]

Table A.2: BNF of Instance Query Expression

Construction	
construct	::= "construct" outMulti
outMulti	::= outBase{(",","/")outBase}
outBase	::= outVariable([allInfo]"["outMulti"]" withOrder) aggregate [name":".]groupings ["outMulti"] outbase ":" outbase
outVariable	::= [name":".][schm]variable
allInfo	::= "["
groupings	::= "{"outBase {"/"outBase}"}
withOrder	::= "order by" ((variable aggregate) ["desc" "asc"] "(" (variable aggregate) ["desc" "asc"] {"," (variable aggregate) ["desc" "asc"]} ")"
aggregate	::= ("count" "avg" "sum") "(" ({"["DISTINCT"] variable"}) ("max" "min") "(" (["DISTINCT"] variable [","NUMBER] ">)"
schm'	::= "(" ["!"]contextName{("," " ") ["!"]contextName}"

Table A.3: BNF of Result Construction Expression

Bibliography

- [1] Serge Abiteboul. Querying Semi-Structured Data. *The International Conference on Database Theory*, 1997.
- [2] Serge Abiteboul and Anthony Bonner. Objects and Views. In *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, pages 238–247, New York, NY, USA, 1991. ACM.
- [3] Serge Abiteboul, Dallon Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.
- [4] Antonio Albano, Roberto Bergamini, Giorgio Ghelli, and Renzo Orsini. An Object Data Model with Roles. In *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases*, pages 39–51, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.
- [5] Antonio Albano, Giorgio Ghelli, and Renzo Orsini. A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language. In *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases*, pages 565–575, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.
- [6] Marc Andries, Marc Gemis, Jan Paredaens, Inge Thyssens, and Jan V. Bussche. Concepts for Graph-Oriented Object Manipulation. In *EDBT '92: Proceedings of the 3rd*

- International Conference on Extending Database Technology*, pages 21–38, London, UK, 1992. Springer-Verlag.
- [7] Renzo Angles and Claudio Gutierrez. Survey of Graph Database Models. *ACM Comput. Surv.*, 40(1):1–39, 2008.
 - [8] Malcolm Atkinson, David DeWitt, David Maier, François Bancilhon, Klaus Dittrich, and Stanley Zdonik. The Object-Oriented Database System Manifesto. pages 1–20, 1992.
 - [9] Michael Benedikt, Wenfei Fan, and Floris Geerts. XPath Satisfiability in the Presence of DTDs. *J. ACM*, 55(2):1–79, 2008.
 - [10] Elisa Bertino and Giovanna Guerrini. Objects with Multiple Most Specific Classes. In *ECOOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 102–126, London, UK, 1995. Springer-Verlag.
 - [11] Tim Bray, Jean Paoli, and Eve Maler. Extensible Markup Language (XML) 1.0 (Fifth Edition). <http://www.w3.org/TR/REC-xml/>.
 - [12] Bin Cao and Antonio Badia. SQL Query Optimization Through Nested Relational Algebra. *ACM Trans. Database Syst.*, 32(3):18, 2007.
 - [13] Dunren Che, Karl Aberer, and M. Tamer Özsu. Query Optimization in XML Structured-Document Databases. *VLDB J.*, 15(3):263–289, 2006.
 - [14] Peter Pin-Shan Chen. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.
 - [15] Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, 1970.
 - [16] Compiler tools.net. The Lex & Yacc Page. <http://dinosaur.compilertools.net/>.

- [17] Oracle Cooperation. Oracle Database SQL Reference 10g Release 1 (10.1). http://stanford.edu/dept/itss/docs/oracle/10g/server.101/b10759/statements_5001.htm.
- [18] Mohamed Dahchour, Alain Pirotte, and Esteban Zimányi. A Generic Role Model for Dynamic Objects. In *CAiSE '02: Proceedings of the 14th International Conference on Advanced Information Systems Engineering*, pages 643–658, London, UK, 2002. Springer-Verlag.
- [19] Mary Fernandez, Ashok Malhotra, Jonathan Marsh, Marton Nagym, and Norman Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM). <http://www.w3.org/TR/xpath-datamodel/>.
- [20] Georg Gottlob, Michael Schrefl, and Brigitte Röck. Extending Object-Oriented Systems with Roles. *ACM Trans. Inf. Syst.*, 14(3):268–296, 1996.
- [21] Marc Gyssens, Jan Paredaens, Jan Van den Bussche, and Dirk Van Gucht. A Graph-Oriented Object Database Model. *IEEE Trans. on Knowl. and Data Eng.*, 6(4):572–586, 1994.
- [22] Sven Hartmann and Sebastian Link. XML Query Optimisation: Specify Your Selectivity. *Database and Expert Systems Applications, International Workshop on*, 0:30–34, 2007.
- [23] Su Cheng Haw and G. S. V. Radha Krishna Rao. Query Optimization Techniques for XML Databases. *International Journal of Information Technology*, 2(2):97–104, 2005.
- [24] Jan Hidders. Typing Graph-Manipulation Operations. In *ICDT '03: Proceedings of the 9th International Conference on Database Theory*, pages 394–409, London, UK, 2002. Springer-Verlag.

- [25] Richard Hull and Roger King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Comput. Surv.*, 19(3):201–260, 1987.
- [26] Hideko S. Kunii. DBMS With Graph Data Model for Knowledge Handling. In *ACM '87: Proceedings of the 1987 Fall Joint Computer Conference on Exploring technology: today and tomorrow*, pages 138–142, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.
- [27] Gabriel M. Kuper. *The Logical Data Model: a New Approach to Database Logic*. PhD thesis, Stanford, CA, USA, 1986.
- [28] John Levine. *Flex & Bison*. O'Reilly Media, 2009.
- [29] Mengchi Liu and Jie Hu. Information Networking Model. *Submitted for journal publication*.
- [30] Mengchi Liu and Jie Hu. Information Networking Model Query Language. *Submitted for journal publication*.
- [31] Mengchi Liu and Jie Hu. Information Networking Model. In *Proceedings of ER*, pages 131–144, Gramdo, Brazil, November 2009.
- [32] Mactech.com. Using Flex and Bison. <http://www.mactech.com/articles/mactech/Vol.16/16.07/UsingFlexandBison/>.
- [33] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallas Quass, and Jennifer Widom. Lore: a Database Management System for Semistructured Data. *SIGMOD Rec.*, 26(3):54–66, 1997.
- [34] Jason McHugh and Jennifer Widom. Query Optimization for XML. *The VLDB Journal*, pages 263–289, 1999.

- [35] Alberto O. Mendelzon, George A. Mihaila, and Tova Milo. Querying the World Wide Web. *Int. J. on Digital Libraries*, 1(1):54–67, 1997.
- [36] Alberto O. Mendelzon and Tova Milo. Formal Models of Web Queries. *Inf. Syst.*, 23(8):615–637, 1998.
- [37] Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *ICDT*, pages 277–295, 1999.
- [38] Oracle.com. Berkeley DB Products. <http://www.oracle.com/technology/products/berkeley-db/index.html>.
- [39] Shankar Pal, Istvan Cseri, Oliver Seeliger, Michael Rys, Gideon Schaller, Wei Yu, Dragan Tomic, Adrian Baras, Brandon Berg, Denis Churin, and Eugene Kogan. XQuery Implementation in a Relational Database System. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1175–1186. VLDB Endowment, 2005.
- [40] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object Exchange Across Heterogeneous Information Sources. In *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260, Washington, DC, USA, 1995. IEEE Computer Society.
- [41] Jan Paredaens and Bart Kuijpers. Data Models and Query Languages for Spatial Databases. *Data Knowl. Eng.*, 25(1-2):29–53, 1998.
- [42] Joan Peckham and Fred Maryanski. Semantic Data Models. *ACM Comput. Surv.*, 20(3):153–189, 1988.
- [43] GNOME Project. GLib. <http://library.gnome.org/devel/glib/2.6/glib.html>.

- [44] Joel Richardson and Peter Schwarz. Aspects: Extending Objects to Support Multiple, Independent Roles. *SIGMOD Rec.*, 20(2):298–307, 1991.
- [45] Friedrich Steimann. On the Representation of Roles in Object-Oriented and Conceptual Modelling. *Data Knowl. Eng.*, 35(1):83–106, 2000.
- [46] Jianwen Su. Dynamic Constraints and Object Migration. In *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases*, pages 233–242, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.
- [47] Raymond K. Wong, H. Lewis Chau, and Frederick H. Lochovsky. A Data Model and Semantics of Objects with Dynamic Roles. In *ICDE '97: Proceedings of the Thirteenth International Conference on Data Engineering*, pages 402–411, Washington, DC, USA, 1997. IEEE Computer Society.
- [48] Zhi Yang and Shiguang Ju. Structural Clustering Storage Method of XML Data Based on SAX. *Computer Engineering*, 34(18):72–74, 2008.