Query Evaluation in Deductive and Object-Oriented Databases

by

Zhaohui Xie

B. Sc., Fudan University, Shanghai, China, 1984

M. Sc., Fudan University, Shanghai, China, 1987

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
in the School
of
Computing Science

© Zhaohui Xie 1995
SIMON FRASER UNIVERSITY
January 1995

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

APPROVAL

Name: Zhaohui Xie

Degree: Doctor of Philosophy

Title of thesis: Query Evaluation in Deductive and Object-Oriented Databases (DOOD)

Examining Committee: Dr. Tom Shermer
Chair

____

Dr. Jiawei Han, Senior Supervisor

____

Dr. Fred Popowich, Supervisor

____

Dr. Veronica Dahl, Supervisor

____

Dr. William S. Havens, Internal Examiner

____

Dr. Jia-Huai You, External Examiner

Date Approved: January 23, 1995

SIMON FRASER UNIVERSITY

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Query Evaluation in Deductive and Object Oriented Databases.

Author: ______________________

(signature)

Zhaohui Xie

(name)

February 9, 1995

(date)

Abstract

The development of deductive and object-oriented database (DOOD) systems by integration of the object-oriented paradigm with the deductive paradigm represents a promising direction in the construction of the next generation of database systems. This thesis addresses the issues of query evaluation in DOOD systems and presents some promising approaches to the problems in DOOD query evaluation.

First, a DOOD data model and its query language are presented to demonstrate the salient features supported by DOOD and to serve as research vehicles for the investigation of DOOD query evaluation. After a comparative survey on query evaluation methods for relational databases, deductive databases and object-oriented databases, the impact of DOOD models and languages on query evaluation is discussed. A list of open and not well-solved problems is identified as a research guide towards DOOD query evaluation, which in turn motivates the research efforts presented in this thesis.

An efficient navigation structure, called the join index hierarchy, is proposed to handle the problem of "pointer-chasing" or "gotos' on disks" in exploring logical relationships among complex objects. Effective optimization strategies are introduced to employ the constraint conditions expressed in the form of complex selection and join conditions for efficient set-oriented navigations and to exploit the common navigations among a query and encapsulated methods for efficient query evaluation. The query-independent compilation and chain-based evaluation, developed for deductive recursive query evaluation, are extended to process a class of DOOD recursions.

Acknowledgments

First of all, I would like to thank my senior supervisor Dr. Jiawei Han for his guidance, help and encouragement while I was conducting the thesis research. This thesis comes from the numerous inspiring discussions with him. I also would like to thank my supervisor Dr. Fred Popowich for his valuable and detailed comments on my proposal and thesis. I am grateful to my supervisor Dr. Veronica Dahl for her constant encouragement. She spent time reading my thesis carefully while she was far away in Europe.

I thank Dr. William Havens and Dr. Jia-Huai You for being my examiners. Their comments and suggestions help to improve the thesis greatly.

I appreciate the help and encouragement from our graduate program director Dr. Lou Hafer and our graduate secretary Kersti Jaager.

Thanks go to my fellow students Mark Mezofeny and Graham Finlayson for going to movies with me so many times! I would also like to thank my fellow students in the database group, Ling Liu, Yongjian Fu, Osmar Zaiane, Kris Koperski, Gabor Melli and Max Luk for the valuable discussions.

Finally, my most heartfelt thanks go to my parents, grandma, brother and cousin for their love, support and encouragement. I am grateful to my dear Shimin for her wonderful love which brightened many days of my academic struggle.

Contents

Abstract iii
Acknowledgments iv
List of Tables ix
List of Figures x
1 Introduction 1
1.1 Motivation 1
1.2 DOOD and Problems in DOOD Query Evaluation 3
1.3 Contributions 5
1.4 Thesis Organization 6
2 Deductive and Object-Oriented Database Model and Language 8
2.1 Introduction 8
2.2 DOOD Data Model 9
2.3 DOOD Query Language 13
3 Query Evaluation in Database Systems 16
3.1 Introduction 16
3.2 Query Evaluation Techniques in Relational Databases 17
3.3 Deductive Query Evaluation Techniques 19
3.4 Query Evaluation in Object-Oriented Databases 21
3.4.1 Graph-Based Object-Oriented Query Optimization . 22
3.4.2 Algebraic Transformation-Based Query Optimizations in Object-Oriented Databases 23
3.5 Problems in DOOD Query Evaluation 25
3.5.1 Navigation through Complex Objects 26

3.5.2 Queries Including Complex Selections, Joins and Aggregations . . . 29
3.5.3 Methods and Encapsulation . . . 30
3.5.4 Recursive Query Evaluation . . . 32
4 Join Index Hierarchies for Efficient Navigations . . . 34
4.1 Introduction . . . 34
4.2 Preliminaries . . . 37
4.3 Construction and Maintenance of Join Index Hierarchies . . . 42
4.3.1 Construction of a Partial Join Index Hierarchy . . . 42
4.3.2 Update Maintenance of a Partial Join Index Hierarchy 48
4.3.3 Base and Complete Join Index Hierarchies . . . 49
4.4 Performance Evaluation of Join Index Hierarchies . . . 50
4.4.1 Storage and Navigation Costs . . . 51
4.4.2 Update Cost . . . 52
4.4.3 Explanation of Performance Results . . . 54
4.5 Discussion . . . 60
4.5.1 Join Index Hierarchy Which Supports Other Kinds of Navigations . . . 60
4.5.2 "Fire Walls" in the Construction of Join Index Hierarchies . . . 63
4.6 Summary . . . 64
5 Optimizing Queries Including Complex Selections, Joins, Aggregations and Methods . . . 65
5.1 Introduction . . . 65
5.2 Motivating Examples . . . 67
5.3 Path Expression Definition . . . 73
5.4 Optimization of Complex Selections and Joins . . . 76
5.4.1 Path Expression-Based Selections . . . 76
5.4.2 Path Expression-Based Joins . . . 79
5.5 Classification and Cost Estimation of Methods . . . 83
5.5.1 Method Definition . . . 83

5.5.2 Method Classification 86
5.5.3 Cost Estimation of Method Evaluation 89
5.6 Query Graph and Query Plan Generation 91
5.6.1 Query Graph 93
5.6.2 Query Plan Generation 99
5.7 Discussion 102
5.7.1 Optimization Strategies for Supporting Other Kinds of Navigations 102
5.7.2 Method Materialization for Query Evaluation 104
5.7.3 Integrating with Indexing Techniques 105
5.7.4 Integrating with Techniques for Searching Optimal Query Evaluation Plan 106
5.8 Summary 107
6 Recursive Query Evaluation 109
6.1 Introduction 109
6.2 Normalization and Classification 112
6.3 Compilation and Evaluation of Linear Recursions 117
6.4 Discussion 128
6.4.1 Compilation and Evaluation of Nested Linear Recursions 128
6.4.2 Integrating with Indexing Techniques 131
6.4.3 Exploring Typing Information 132
6.5 Summary 133
7 Conclusion and Future Research 134
7.1 Summary 134
7.2 Future Research 135
7.2.1 Indexing over Class Hierarchy 135
7.2.2 Extensibility to Support Abstract Data Types and Search Strategies 136
7.3 Concluding Remarks 138
Appendices
A Evaluation of Some Parameters in Chapter 4 140

B Sample Database 142
C Proof Sketch of Theorem 5.1 and Theorem 5.5 145
D Unification Definition 148
E Proof Sketch of Theorem 6.1 150
Bibliography 152

List of Tables

4.1 Database Parameters 50

4.2 Database Parameter Values 51

List of Figures

4.1 Navigation. 35

4.2 A Schema Path of Length 5. 38

4.3 Three Kinds of Join Index Hierarchies Corresponding to the Schema Path in Figure 4.2 40

4.4 Two Partial Join Index Hierarchy Structures for Supporting JI(0,4) and JI(2,5). 44

4.5 Build a Partial Join Index Hierarchy and Propagate Update. 46

4.6 Storage Costs of B-JIH, P-JIH, C-JIH and Full-ASR vs. Fan-outs. . . 54

4.7 Navigation Costs of B-JIH, P-JIH, C-JIH and Full-ASR vs. Fan-outs. 55

4.8 Update Costs of B-JIH, P-JIH, C-JIH and Full-ASR vs. Fan-outs. . . 56

4.9 Costs of Navigation and Update mix for B-JIH, P-JIH, C-JIH and Full-ASR. 57

4.10 Navigation Costs of B-JIH, P-JIH, C-JIH and Full-ASR vs. Navigation Selectivities. 57

4.11 Storage Explosion with Large Fan-outs. 58

4.12 Partial Join Index for Supporting JI(0,5). 58

4.13 Associative Search Costs of B-JIH, P-JIH, C-JIH, Full-ASR and Nested Index vs. Fan-outs. 59

4.14 Update Costs of B-JIH, P-JIH, C-JIH, Full-ASR and Nested Index vs. Fan-outs. 60

5.1 Query Graph of Example 5.10 96

5.2 Query Graph of Example 5.11 97

5.3 Query Plan Generation of Example 5.10 100
5.4 Query Plan Generation of Example 5.11 101

Chapter 1

Introduction

1.1 Motivation

Effective and efficient management of large volumes of complex data are required in the advanced applications, such as computer-aided design and manufacturing, multimedia applications with audio and video data, and scientific and medical applications. New generation of database systems needs to support high level languages for defining, reasoning, retrieving and manipulating the complex data and to provide efficient software architectures and techniques for achieving greater modeling power and higher performance.

Research towards deductive database systems represents a promising direction in declarative database programming and integration of logic programming paradigm and relational database technology. Deductive database systems have made great strides in recent years with encouraging progress in fundamental research and implementation [121, 104, 105]. A large number of research prototypes have been developed to date, e.g., Aditi [122], CORAL [103], DECLARE and SDS [72], EKS [127], Glue-Nail! [33], LDL [27], LogicBase [50, 52], LOLA [36], MegaLog [17], and XSB [107]. These achievements clearly show that the deductive database technology has reached a level of maturity so that the development of deductive database systems for real applications is feasible.

On the other hand, research towards object-oriented and extensible database systems represents a promising direction in extending relational database technology and integrating object-oriented programming paradigm and database technology for supporting a rich collection of sophisticated data modeling and manipulation concepts [4, 6, 81, 112, 115]. A large number of research prototypes and commercial systems have been available, e.g., DASDBS [108], EXODUS and Volcano [40, 19, 42], GemStone [18], GENESIS [9], Iris and OpenODB [35], O2 [5], Objectivity/DB [98], ObjectStore [88], Ode [1], Ontos [99], Orion and Itasca [80], Postgres and Montage [116], Probe [100], Starburst [45, 54], UniSQL [82], VERSANT [125], Zeitgeist and OpenOODB [128]. The intensive research and commercialization activities demonstrate that object-oriented database systems are at the forefront of supporting new applications.

Although deductive database systems and object-oriented database systems have emerged as promising approaches for supporting advanced applications, both of them suffer from some drawbacks [10, 120, 75]. Despite a highly declarative and powerful logical framework, deductive databases only support a flat data model and do not provide a very powerful modeling mechanism. Object-oriented databases do not have a recognized logical foundation, which traditionally was considered as very important for database programming languages, and do not provide a highly declarative language interface for accessing and manipulating complex data. There is growing consensus that integrating the object-oriented paradigm and rule-based deduction may provide a powerful framework for the next generation of database systems, the so-called deductive and object-oriented databases (DOOD) [1].

DOOD models and languages address the issues of enhancing data modeling power and high declarativeness of database languages for advanced applications. The high complexity of DOOD models and languages requires efficient software architectures and techniques so that DOOD systems can achieve competitive or higher performance than those of traditional database applications. Efficient query evaluation has been crucial to the success of relational database systems. Thus it is expected that the

[1]The declarativeness of deductive database languages is so emphasized that deductive and object-oriented databases are also called *declarative and object-oriented databases* [104].

success of DOOD will largely rely on the system performance, especially on the efficient query evaluation techniques over large volumes of complex data. This thesis will concentrate on DOOD query evaluation.

1.2 DOOD and Problems in DOOD Query Evaluation

DOOD languages combine deductive database languages and object-oriented database languages and support features such as declarativeness, deduction, recursion, complex objects, object identities, class hierarchies, inheritances, encapsulated methods and abstract data types. A query, expressed in a DOOD language, is only a specification of what a user wants but not how a user's want can be executed. Thus the DOOD query language is said to be declarative rather than procedural.

Query evaluation is a process to find and perform an efficient execution of a declarative query for information from databases. The goal of a query optimization is to translate a query into an (sub)optimal evaluation plan for accessing and manipulating the databases. A query execution engine is then responsible for executing the optimal plans. Relational query evaluation techniques have been successful in the optimization of declarative query languages [112]. Extensive investigations have been conducted on those key techniques, such as algebraic query optimization, join methods, and search strategies for optimal query evaluation plans [62, 95, 39]. Recent research into deductive query evaluation has made significant progress in recursive query optimization [104, 105]. Some of the techniques outperform non-recursive (relational) query optimization in commercial database systems [104]. Query evaluation in object-oriented database systems is still developing and is the focus of the recent intensive research [6, 81]. Some relational query evaluation techniques, such as algebraic query optimization and join methods, have been extended to query evaluation in object-oriented database systems.

Although there are some similarities between a relational database language and a DOOD language, the new DOOD features require new evaluation techniques to efficiently process complex DOOD queries. The traditional query evaluation techniques in relational databases and deductive databases are not powerful enough to handle

all kinds of complex queries. The following is a list of some recognized open or not well-solved problems in DOOD query evaluation:

- *Support for efficient navigation or "pointer-chasing".* Navigation is an essential logical operation in DOODs. A navigation is performed following object identifiers along a sequence of logical connections among complex objects. These connections represent logical relationships specified by class/subclass relationships, attribute relationships, methods, and deduction rules. Navigation operations may cause significant performance suffering because objects along a navigation path may be scattered in different pages or blocks of disks, and many I/O operations on disks may be required to load those related objects into main memory.

- *Optimization of queries in the presence of complex selections, joins and aggregations.* Path expressions, the main syntactic notions in DOOD languages, are frequently used to express navigations over complex object structures. Navigations are performed over database objects, but are confined to selective objects in databases. The constraints are presented in the form of selection and join conditions, possibly with aggregation functions. These selections and joins are certainly more general and complicated than those in relational database systems. The optimization of queries including these complex selections and joins is not well investigated. This issue is clearly related to the previous one.

- *User-defined methods and encapsulation.* The optimization of queries including encapsulated methods is still an open problem. Encapsulation is an effective mechanism for software maintenance. However, it blocks various kinds of the information, such as the semantics of a method, the cost of computing a method, the output size of applying a method to a set of input objects, the implementation of a method, and the navigation operations in a method. These kinds of the information are considered very important for efficient query processing.

- *Recursive queries.* Recursive query evaluation was the focus of intensive research in deductive databases. However, little has been done on DOOD recursive query evaluation. It is not clear how the new features of DOOD impact the recursive

query evaluation and how the recursive query evaluation techniques, developed for deductive database systems, can be extended to handle DOOD recursive query evaluation.

This list does attempt to serve as a guide for our research towards DOOD query evaluation. However, it is, by no means, a complete list of all open or not-well solved problems. There are other interesting problems in DOOD query evaluation. For example, the full support of object identities may lead to several definitions of "equivalent" between "semantically equivalent" object algebraic expressions. This causes not only the various problems of view definition in object-oriented database systems but also the equivalence conservation problem when performing algebraic query optimization [101, 110].

An object in a class can inherit attributes and methods from its superclasses. The implementations of these attributes and methods, however, could be different from those of its superclasses. It is impossible, at query compilation and/or optimization time, for a query optimizer to determine which implementations would be invoked at query run time. The polymorphism of attribute names and methods, therefore, makes it difficult for the query optimizer to perform optimization before queries are actually evaluated.

1.3 Contributions

This thesis focuses on DOOD query evaluation and presents new approaches to the issues in the previous list. The following research results constitute the main contributions of this thesis.

- *Support for efficient navigations* [130, 131]. A novel indexing structure, called the *join index hierarchy*, is proposed to handle the problem of "goto's on disk" or navigations through complex objects. The method constructs a hierarchy of join indices and transforms a sequence of pointer chasing operations into a simple search in an appropriate join index file, and thus accelerates navigations. The method extends the join index structure studied in relational and spatial

databases, supports both forward and backward navigations among objects and classes, and localizes update propagations in the hierarchy. An experimental study shows that partial join index hierarchy outperforms several other indexing mechanisms in overall performance.

- *Query optimization in the presence of complex selections, joins, aggregations and encapsulated methods* [129, 132]. A systematic classification of complex selections and joins is presented. It is illustrated that different types of selections and joins require different kinds of optimization strategies, and some selections and joins with aggregation functions can be transformed into equivalent but more efficient forms of selections and joins without aggregation functions. Path expressions (navigation operations) in encapsulated methods are revealed. Consequently, common navigation operations among encapsulated methods and queries can be exploited for improving query evaluation efficiency. Query graphs are employed to generate query evaluation plans.

- *Recursive query compilation and evaluation* [50, 51, 52]. The influence of DOOD features on recursive query evaluation is investigated. A normalization process is proposed to serve not only as a pre-processing stage for compilation and evaluation but also as a tool for classifying recursions. A class of recursions, called DOOD linear recursion, is identified which can be efficiently processed by the extension of the query-independent compilation and chain-based evaluation. In addition, the evaluation of DOOD nested linear recursions and the integration with other evaluation techniques are discussed as well.

1.4 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 describes a DOOD model and a query language. The model and language are based upon F-logic [75] and XSQL [73]. They serve as research vehicles for the investigation of DOOD query evaluation in the later chapters. In Chapter 3, a comprehensive review of query evaluation techniques in relational, deductive and object-oriented database systems is presented. Problems

beyond relational query evaluation techniques are discussed. These problems motivate the research efforts reported in this thesis. In Chapter 4, a new index structure, called the join index hierarchy, is proposed to handle the problem of navigations through complex objects. Chapter 5 presents a systematic study of query optimization in the presence of complex selections, joins, aggregations and encapsulated methods. In Chapter 6, the influence of DOOD features on recursive query evaluation is investigated. The query-independent compilation and chain-based evaluation are extended to handle a class of DOOD recursive queries. Finally, Chapter 7 concludes the thesis with discussion and future research.

Chapter 2

Deductive and Object-Oriented Database Model and Language

2.1 Introduction

Considerable efforts have been made towards integrating a logic paradigm with object-orientation and formalizing the concepts of object-orientation, e.g., [2, 3, 93, 26, 74, 76, 25, 75, 87, 75]. Maier's O-logic [93], influenced by Aït-Kaci's work on ψ-terms [2, 3], represents an early attempt. The subsequent work on C-logic [26], F-logic [74, 76] and PathLog [37] continues this effort. Although Hilog [25] is itself not a database programming language, it provides an implementation platform for object-oriented languages. SchemaLog [87] extends the approaches of Hilog and F-logic and provides a logical foundation of schema integration and evolution in heterogeneous database systems. Several database research prototypes, for example, CORAL++ [114], LDL++, and COMPLEX [43], incorporate some limited object-oriented features into deductive database languages.

In this chapter, a DOOD data model based upon F-logic [75] and a DOOD query language based upon XSQL [73] are introduced not only to demonstrate the salient features of a DOOD model and language but also to serve as research vehicles for the investigation of DOOD query evaluation in the later chapters.

F-logic is a logic with higher-order syntax[1]. It supports all the major features of object-orientation including those higher-order concepts such as objects, object identities, classes, methods, class hierarchies, inheritance, etc. Its semantics is restricted enough so that the first-orderness is preserved. F-logic intends to stand in the same relationship to the object-oriented paradigm as the first-order predicate calculus to the relational system, and to lay a logical foundation for object-oriented data model.

2.2 DOOD Data Model

Like predicate calculus, the language of F-logic consists of a set of formulas which are built from object molecules connected by $\neg$, $\vee$ and $\wedge$, and the quantifiers $\exists$ and $\forall$, and object molecules which are built from id-terms.

Definition 2.1 id-term. An id-term is defined inductively:

- a variable is an id-term.
- a constant is an id-term.
- $f(t_1, ..., t_n)$ is an id-term if f is an n-ary function and $t_1, ..., t_n$ are id-terms.

Objects are referenced via their denotations or identifiers which are ground id-terms. For example, $alex$ and $mother(alex)$ are ground id-terms which act as object identifiers. An id-term can alos be used to represent a class name, attribute name and a method name.

Definition 2.2 Object molecules. There are four kinds of object molecules:

- Predicate object molecules: any predicates are predicate object molecules.

[1]By first-order syntax, we mean that variables cannot appear in the places where predicates and/or function symbols do. In a logic with a higher-order syntax, variables are allowed to appear in the places where predicates and/or function symbols do. In a logic with a first-order semantics, variables can only range over domains of individuals or over the names of the predicates and functions. By higher-order semantics, we mean that variables can range over domains of relations and functions constructed from the domains of individuals [25].

- is-a object molecules: either P:Q or P::Q where P and Q are id-terms. The first denotes a class-membership while the second represents a class/subclass relationship.

- data object molecules, denoted by

  $$C[l_1 \to C_1; ...; l_n \to C_n],$$

  or

  $$C : D[l_1 \to C_1; ...; l_n \to C_n],$$

  where C is an id-term representing an object identity and D is an id-term denoting a class which C belongs to. A label l_i is of the form *Attr* or $Method@X_1, ..., X_m$ where *Attr* is an id-term representing an attribute name, and *Method* and X_i are id-terms representing a method name and argument objects respectively. C_i is an id-term, a set of id-terms, a data object molecule or a set of data object molecules representing the method result.

- signature object molecules, denoted by

  $$C[l_1 \Rightarrow C_1; ...; l_n \Rightarrow C_n],$$

  where C_i is an id-term representing a class name. A label l_i is of the form *Attr* or $Method@X_1, ..., X_m$ where *Attr* is an id-term representing an attribute name, and *Method* and X_i are id-terms representing a method name and argument object types respectively. C_i is an id-term or a set of id-terms representing the method result type.

For example, $STUDENT :: PERSON$ is an is-a object molecule. It denotes that $STUDENT$ is a subclass of $PERSON$. $alex : STUDENT$ is also an is-a object molecule which represents that $alex$ is an instance of the class $STUDENT$.

Example 2.1 Data object molecule for the student *alex*.

$$
\begin{array}{rcrcrcr}
alex[Name & \rightarrow & \text{“}Alex\text{”}; & & & & \\
Age & \rightarrow & 30; & & & & \\
HomeAddress & \rightarrow & addr[Country & \rightarrow & canada[Name & \rightarrow & \text{“}Canada\text{”}; \\
 & & & & Population & \rightarrow & 24,000,000; \\
 & & & & Area & \rightarrow & 9,900,000]; \\
 & & City & \rightarrow & vancouver[Name & \rightarrow & \text{“}Vancouver\text{”}; \\
 & & & & Population & \rightarrow & 1,500,000; \\
 & & & & Area & \rightarrow & 500]; \\
 & & StreetName & \rightarrow & \text{“}Winch\text{”}; & & \\
 & & StreetNumber & \rightarrow & 6000; & & \\
 & & ZipCode & \rightarrow & V5B\ \ 2L3]; & & \\
GPA & \rightarrow & 3.8; & & & & \\
TakeCourse & \rightarrow & \{cmpt400\}]. & & & &
\end{array}
$$

It defines several attributes of *alex*, i.e., *Name*, *Age*, *HomeAddress*, etc. The following is also a data object molecule which defines the values of applying method

$$FromLargeCountryOrMetro$$

to *alex* to be *True*,

$$alex[FromLargeCountryOrMetro@ \rightarrow True].$$

□

Example 2.2 Signature object molecule for class *DEPT*

$$DEPT[Name \Rightarrow String; Chairperson \Rightarrow PROF; OfferCourse \Rightarrow \{COURSE\}].$$

Signature object molecules are used to express the typing constraints. For example, the *Name* of a department is of type *String* and the *Chairperson* of a department is of type *PROF*. □

Definition 2.3 Formulas. Formulas can be defined inductively:

•• Object molecules are formulas;

•• $\varphi \vee \psi$, $\varphi \wedge \psi$ and $\neg\varphi$ are formulas if φ and ψ are formulas;

•• $\forall X \varphi$, $\exists Y \psi$ are formulas if φ and ψ are formulas, and X and Y are variables.

Definition 2.4 Rule. A rule has the form

$$H \longleftarrow B_1, ..., B_n,$$

where H[2] and B_i are object molecules.

Example 2.3 Rule.

$$X[StudentOf@ \rightarrow \{Y\}] \longleftarrow$$

$$Y : PROF, Y[TeachCourse \rightarrow \{C\}], X : STUDENT[TakeCourse \rightarrow \{C\}].$$

If a student is taking a course taught by a professor, then the student is a student of the professor. ◫

Definition 2.5 Query. A query has the same form as a rule without a head.

Example 2.4 Query.

$$\begin{array}{ll} ?- & S : STUDENT[\\ & HomeAddress \rightarrow addr[Country \rightarrow country[Name \rightarrow \text{“}Canada\text{”}]]; \\ & TakeCourse \rightarrow \{courses[Dept \rightarrow cs[Name \rightarrow \text{“}Computer\ \ Science\text{”}]]\}; \\ & GPA \rightarrow 4.0]. \end{array}$$

The query is to find all the students who come from Canada, are taking some computer science courses and have a 4.0 GPA. ◫

The above query could be presented in a simpler form by using a path expression. A path expression, a syntactic notion, is often used to refer complex objects. Intuitively, a path expression is a traversal over the class composition hierarchy. For example,

$$s.HomeAddress.Country.Name$$

[2] H is a normalized object molecule. The definition of a normalized object molecule is in Chapter 6

describes a path expression which starts from an object s in class $STUDENT$, continues to the home address and the country of s, and ends at the name of the country. It simply denotes the name of the country of the student s. The value of a path expression can be either single-valued or set-valued which depends on whether there are set-valued attributes in the path expression. For example, the value of the following path expression

$$s.TakeCourse.Dept.Name$$

is a set of the names of the departments which offer the courses the student s is taking. Here $TakeCourse$ is a set-valued attribute. Instead of

$$S[HomeAddress \rightarrow addr[Country \rightarrow country[Name \rightarrow \text{“}Canada\text{”}]]],$$

it could be written as the following path expression

$$S.HomeAddress.Country.Name = \text{“}Canada\text{”}.$$

Therefore the above query could be rewritten as

$$\begin{aligned} ?-S:STUDENT,\quad & S[S.HomeAddress.Country.Name = \text{“}Canada\text{”}, \\ & S.TakeCourse.Dept.Name = \text{“}Computer\ \ Science\text{”}, \\ & S.GPA = 4.0. \end{aligned}$$

2.3 DOOD Query Language

XSQL [73] is a query language based on F-logic and incorporates all major object-oriented features. It is capable of expressing sophisticated queries in a very compact way via path expressions. In XSQL, selections and joins could be in a more general form than those in relational databases. For example,

$$s.HomeAddress.Country.Name = \text{“}Canada\text{”}$$

selects all the students s who are from Canada. And

$$s.TakeCourse.Dept.Name_{\forall} = \text{“}Computer\ \ Science\text{”}$$

selects all the students s who are only taking computer science courses. Here $\forall$ is a quantifier over the set which is the value of the path expression $s.TakeCourse.Dept.Name$. The comparisons between path expressions form joins. A join between STUDENT and PROF could have the following comparison

$$s.TakeCourse_{\forall} = p.TeachCourse_{\exists}$$

or

$$s.TakeCourse \subseteq p.TeachCourse$$

as a join predicate. It describes a pair of a student and a professor such that all the courses taken by the student are taught by the professor.

Example 2.5 Finds all pairs of students and universities such that all the faculty members of the university are over 40, the total numbers of the courses offered by the universities are over 500, all the courses taken by the students are taught by the university presidents, and the students are from the city Prince George.

$$
\begin{array}{ll}
SELECT & u.Name,\ s.Name \\
FROM & u\ UNIVERSITY,\ s\ STUDENT \\
WHERE & u.Depts.FacultyMembers.Age_{\forall} > 40 \\
AND & COUNT(u.Depts.OfferCourse) > 500 \\
AND & s.TakeCourse_{\forall} = u.President.TeachCourse_{\exists} \\
AND & s.HomeAddress.City.Name = \text{“}Prince\ \ George\text{”}
\end{array}
$$

□

Example 2.6 Finds all the names of students who are from a metropolis or a large country with a population over 20,000,000, and who are only taking computer science courses and taking courses over 400 level.

$$
\begin{array}{ll}
SELECT & s.Name \\
FROM & s\ STUDENT \\
WHERE & FromLargeCountryOrMetro(s) \\
AND & s.HomeAddress.Country.Population > 20,000,000 \\
AND & s.TakeCourse.Dept.Name_{\forall} = \text{“}Computer\ \ Science\text{”} \\
AND & s.TakeCourse.Number_{\forall} > 400
\end{array}
$$

□

XSQL can be used to query not only databases but also schema information such as class hierarchy and class composition hierarchy information.

Example 2.7 Query schema. Finds all the subclasses of PERSON.

$$\begin{array}{ll} SELECT & X \\ WHERE & X \text{ is-subclass-of } PERSON \end{array}$$

The query returns the names of all subclasses of *PERSON*. □

Example 2.8 Query data and schema.

$$\begin{array}{ll} SELECT & Y \\ FROM & x\ PERSON \\ WHERE & x.Y.City.Name = \text{“}Vancouver\text{”} \end{array}$$

The query returns the attribute name *HomeAddress* if there exists a person in *PERSON* who is from Vancouver. Otherwise it returns *NIL* even though *HomeAddress* is an attribute of *PERSON*. □

The semantics of XSQL is rooted in F-logic as it has been shown in [73] that there exists an effective procedure that for any given XSQL query returns an equivalent first-order query in F-logic.

Chapter 3

Query Evaluation in Database Systems

3.1 Introduction

Database models and languages have significant impacts on query evaluation techniques. Before the introduction of relational database systems, there were two popular approaches used to construct database management systems. The first approach (called hierarchical), exemplified by IBM's IMS [58], has a tree-based data model and navigational query language. All data records are arranged into a collection of trees. An application programmer could navigate from root records to some child records, and access records one at a time by employing the navigational query language. The second approach (called network) is based upon a graph-based data model which was proposed by the Conference on Data Systems Languages (CODASYL) [30]. Similarly, all data records are assembled into a collection of directed graphs. An application programmer could access records from an entry point by navigations over the graphs with a navigational query language. In both approaches, an application programmer must write a complex program to navigate through a database in order to answer a specific database query. It is the responsibility of a user to specify not only what she or he wants but also how her or his wants could be obtained.

In contrast, the relational data model provides a fundamentally different approach.

Data records are represented by relations. A user accesses a database through a declarative query language by specifying what she or he wants. It is the responsibility of a database system to actually evaluate the user's query. A query processor in a database system translates a user's query into an efficient evaluation plan, and executes the plan for accessing or manipulating the database. A query optimizer is responsible for translating a query into an (sub)optimal evaluation plan. The translation, called query optimization, is a process of planning and searching for optimal query evaluation plans and employing techniques from many disciplines, such as artificial intelligence, dynamic programming and operational research. An evaluation plan is optimal in the sense that it minimizes some measures, such as users' waiting time for results produced by a database system, CPU, I/O and network communication time and effort, total resource usage, and possibly the combination of the above measures. After an optimal query evaluation plan is obtained, a query execution engine executes the plan, and accesses or manipulates the database as instructed in the user's query[1].

Query evaluation has been explored in the context of relational database systems and deductive database systems and has received growing attention in the research community for the new generation of database systems. DOOD challenges the traditional query evaluation techniques with its sophisticated modeling power and highly declarative language. The new features, such as complex objects, object identities, classes, methods, encapsulations, inheritances, etc., exert essential influence on query evaluation. In this chapter, query evaluation techniques in relational databases, deductive databases and object-oriented databases are surveyed. The emphasis is on query evaluation in object-oriented databases and the important problems in DOOD query evaluation.

3.2 Query Evaluation Techniques in Relational Databases

Many important studies on relational query evaluation are focused on algebraic query optimization, join methods and strategies of choosing optimal evaluation orders of

[1] In a distributed heterogeneous database system, information may need to be collected from several databases in different locations or of different types.

joins [62, 39].

Algebraic query optimization is a process of transforming an algebraic expression of a user's query into an equivalent and but more efficient form. The new expression could be either logically equivalent to the original one based upon the relational algebra [62, 119, 34, 86], or semantically equivalent to the original expression [83, 21, 22, 134]. The latter transformation is often called semantic query optimization which employs semantic information, e.g., integrity constraints, in databases.

The join is the most important operator in relational databases. In a two-way join, the join result is a subset of the Cartesian product of the two input relations. The elements in the subset satisfy the join condition. Many join methods have been developed and the most important among them are the nested-loops join[79], sorted-merge join [85, 38], hash-based join [84, 95] and pointer-based join methods [111]. Some of the join methods, such as the nested-loops join and pointer-based join, can be applied to deductive and object-oriented databases [111]. The recent surveys [95, 39] cover some comprehensive reviews on join methods.

Most queries in relational databases can be considered as select-project-join queries. Thus selecting an optimal query evaluation plan is to choose an optimal evaluation order of joins. In large database systems, there could be a large number of joins in a complex query. An exhaustive search for an optimal evaluation plan is unacceptable. The pioneer relational database system prototype, System R developed at IBM, proposed the most influential principle of relational query optimization: perform projection and selection as early as possible [109]. The idea is to eliminate data (tuples) irrelevant to answers in an early stage of evaluation so that query execution is accelerated. In addition, many query evaluation plans, which may be quite expensive, are eliminated from consideration. Consequently, the time spent on searching optimal evaluation plans is also reduced. Recently, randomized algorithms such as iterative improvement, simulated annealing and two-phase optimization [61, 118, 59, 60] were introduced to search optimal evaluation plans. An evaluation plan is represented as a join processing tree, which, in turn, can be considered a state in a state space. Randomized algorithms perform random walks in a state space via a series of moves (manipulations of join processing trees). A cost is associated with each state. The

algorithms try to find a state with local (global) minimum cost according to predetermined criteria (termination conditions).

3.3 Deductive Query Evaluation Techniques

The magic-sets technique is the most influential evaluation method for recursive query evaluation in deductive database systems. It was originally proposed by Bancilhon, Maier, Sagiv and Ullman [7], and extended by Beeri and Ramakrishnan [12, 13] later.

The motivation of the magic-sets rule rewriting technique [119, 104] is that a query often only asks for a small set of the entire relation corresponding to an intensional predicate. A top-down search would start from the query as a goal and employ the rules from heads to bodies to create only goals relevant to the query. Some of them, however, may lead to the "dead end". The disadvantage is that this kind of approach may get into recursive loops, perform repeated computation of some subgoals, and is very hard to determine whether all the answers to the query have been found.

On the other hand, a bottom-up search begins from the rule bodies to the heads. It may consider some facts which are not relevant to the query and would not be considered in the top-down approach. However, this approach has advantages because it avoids the problems of looping and repeated computation and facilitates more efficient set-at-a-time processing than tuple-at-a-time processing in the top-down search.

The magic-sets focus on the top-down approach combined with the looping-freedom, easy termination, and the efficient evaluation of bottom-up search. It can be used to rewrite the rules and pass the bindings from the query so that the advantages of both top-down and bottom-up approaches are integrated.

Example 3.1 Same generation. Two people are the same generation if they are siblings or their parents are the same generation. Here "sg" stands for "same generation". It is an intensional predicate. Both "sibling" and "parent" are extensional predicates. The first rule says that if X and Y are siblings then they are the same generation. The second rule means that if the parent U of X and the parent V of Y are the same generation then X and Y are the same generation. The query ?-sg(john,

Z) tries to find all the people who are the same generation as "john".

$$\begin{aligned} sg(X,Y) &:- \ sibling(X,Y). \\ sg(X,Y) &:- \ parent(X,U), sg(U,V), parent(Y,V). \\ &?- \ sg(john,Z). \end{aligned}$$

In all the relevant tuples (c,d), where c and d are the same generation, c must be an ancestor of $john$. A "magic predicate" is used as a filter to retain only those potentially relevant elements. $john$ is the first relevant one, i.e.,

$$magic_sg(john).$$

In a top-down evaluation, if the rule head is of the form $sg(c,Y)$, where c is an ancestor of $john$, then the body of the above recursive rule will be

$$parent(c,U), sg(U,V), parent(Y,V).$$

U must be bound to a parent of c, thus an ancestor of $john$. Therefore, if c is relevant, so are c's parents, i.e.,

$$magic_sg(U) \ :- \ magic_sg(X), parent(X,U).$$

The original rules and the query can thus be rewritten as

$$\begin{aligned} sg(X,Y) &:- \ magic_sg(X), sibling(X,Y). \\ sg(X,Y) &:- \ magic_sg(X), parent(X,U), sg(U,V), parent(Y,V). \\ magic_sg(U) &:- \ magic_sg(X), parent(X,U). \\ magic_sg(john). & \end{aligned}$$

$magic_sg$ simulates how the goals are generated in a top-down evaluation. $magic_sg(john)$ and the generated facts of $magic_sg$ are used as a filter in the rules defining sg to avoid irrelevant facts to the answers. Thus a bottom-up evaluation of the rewritten rules reaches a selective search similar to that achieved by top-down evaluation of the original rules. □

The magic-sets were originally proposed to handle recursive queries, however, it is applicable to non-recursive queries as well and found to be superior to techniques used in commercial database systems [97].

There are other approaches which have the same effects as the magic-sets. For example, the query-subquery approach [126, 127] combines top-down, bottom-up and side-way information passing features. Queries and subqueries are passed top-down from the head of a rule to its body. Answers generated from a rule are returned bottom-up to their corresponding subqueries. In a rule, answers to the first several subqueries are passed side-way to derive answers to the rest of the subqueries. An interesting project XSB led by Warren [25, 107] is founded on SLG resolution, a variant of OLDT resolution, which is top-down evaluation with memoing. SLG resolution is a partial deduction framework. Each query is transformed step by step into a set of answer clauses. It allows arbitrary control strategies for selecting which transformations to apply and has good termination characteristics.

Chain-based evaluation by Han [47, 46, 48] and prototyped by Han, Ling and Xie [50, 52] represents an alternative approach to efficiently evaluate a very popular class of recursive queries. The method is motivated by the observations that most of studied recursions can be compiled into highly regular chain-forms. The compilation can capture the bindings which could be hard to be captured by other methods. The chain-based methods can explore the query constraints, regularity of recursions and other features of a program to efficiently evaluate recursive queries.

3.4 Query Evaluation in Object-Oriented Databases

Although relational data models only support small subset of functionalities of object-oriented data models, the commonalities promise the possibility of employing the relational query optimization and evaluation techniques to object-oriented query processing. Many query optimization techniques in object-oriented database systems are based upon algebras. These algebras are similar to the relational algebra in supporting bulk data types, e.g., sets, but are generalized to support operations on lists, arrays, user-defined operations and inheritance. This indicates that the relational algebraic query optimization techniques can be extended to object-oriented query processing.

The approaches can be classified into two classes: graph-based object-oriented

query optimization and algebraic transformation-based object-oriented query optimization. In the first approach, queries are represented by query graphs. Transformations of queries is performed by manipulating the corresponding query graphs. Different ways of manipulating query graphs lead to different query evaluation plans. The second approach is featured by direct manipulation of queries in algebraic expression forms. The algebraic expressions are translated into equivalent and but more efficient forms for processing.

Any algebraic expressions can be represented in graph forms and vice versa. Thus there is no fundamental difference between the two. In fact, the approach in Cluet and Delobel [29] takes advantages of both. In Lanzelotte et al. [90], the regulations of generating processing trees, i.e., query evaluation plans, are expressed in rewrite rules, and the optimization is similar to the algebraic transformation-based approach.

3.4.1 Graph-Based Object-Oriented Query Optimization

Banerjee et al. [8] propose a very primitive query graph model which resembles the relational one. In a query graph, classes in a query are denoted by nodes, whereas attribute relationships, and classes/subclasses relationships are denoted by edges. The query optimization is considered as selecting an optimal order in which the classes in a query are traversed. Two basic ways of traversing the nodes in a query graph are proposed: forward traversal and backward traversal. A query cost model, which is quite similar to the relational one, is proposed to determine an optimal access plan for a query. Whenever the ordering of nodes in a query graph, i.e., an access plan, has been given, the evaluation of the query is performed in either forward or backward way according to the plan. This approach only considers two traversals, i.e., forward and backward in evaluating queries. Consequently, some better query evaluation plans might be eliminated from consideration.

Lanzelotte et al. [90] consider a more sophisticated graph-based model which captures not only logical relationships expressed in a query but also storage information relevant to the query. A query graph consists of predicate nodes, name trees and dataflow arcs which connect name trees to predicate nodes. The predicate nodes,

which connect name trees and represent explicit joins, contain conjunctive boolean formula of classes and their attributes in the name trees. An input name tree consists of nodes, which denote classes and their attribute classes. The nodes are connected by edges representing the attribute relationships (implicit joins). An output name tree specifies the projection result classes and their attributes in a query. A conceptual schema captures logical relationships between classes in a database while a physical schema captures storage information, such as clustering and path indices. For example, if objects in some classes are clustered together, then there will be one node in the physical schema which represent the cluster. A query is first translated onto the physical schema. The result of the translation is a connection graph which makes explicit the clustering and path indices information related to the query. A query evaluation plan can be represented as a processing tree derived from the connection graph. The selection of optimal query evaluation plans is formulated as a search problem in a processing tree-based space. Rewrite rules for manipulating processing trees are presented for deterministic search to select optimal processing trees.

Cluet and Delobel [29] propose a typed algebra and take advantage of both algebraic rewrite rule and graph-based approaches. The rewrite rules in their approach are encoded with some object storage information, such as path indices and object placement policy. The typed algebra facilitates factorizations of not only local common subexpressions but also global common subexpressions. The algebraic expressions of queries can also be represented by directed acyclic graphs (DAGs) which can capture both logical relationships of classes (types) in queries and physical storage information.

3.4.2 Algebraic Transformation-Based Query Optimizations in Object-Oriented Databases

Shaw and Zdonik [110] propose an object algebra which synthesizes relational concepts with object-oriented databases. Algebra operations include select, project, join, nest, unnest, dupeliminate and coalesce, which are the generalizations of relational operations. The last two are specifically for eliminating duplicate copies of objects

or duplications in tuple attribute values. The algebra model fully supports object identity in the sense that any query results are also considered as objects. Concepts of equality and identity between algebraic expressions are introduced with algebraic transformation rules which preserve the equality or identity between algebraic expressions before and after transformations. The approach adopted in Osborn [101] is also a typical example of an extension of relational algebraic query optimization. An object algebra is proposed to support most features of a structural object model. Transformation rules include idempotence of unary operators, commutativity of unary operators, commutativity of binary operators, associativity of binary operators, and distributivity of unary operators over combine, union, intersect and subtract. Transformation rules, which preserve equality and/or identity of algebraic expressions, are presented as well. These transformation rules form the basis for algebraic query optimizations. Vandenberg and DeWitt [124] present a many-sorted algebra with algebraic operators on grouping, arrays, references and multisets, and a comprehensive collection of transformation rules governing these operators. Optimization of queries including methods and polymorphism of methods and attribute names are also discussed.

Beeri and Kornatzky [11] propose an object-oriented query language by extending a functional programming language. The query language is structured around a small number of bulk data processing abstractions with a set of transformation rules, e.g., condition and composition of function applications, production and construction, apply-to-all, pump, etc. In addition, user-defined methods coded in the same query language are considered as subqueries when a query including the methods are optimized. Two rules, i.e., pushing an apply-to-all into a method and pulling a filter out of a method, are proposed for optimizing a query in the presence of methods.

Object-oriented query optimization in Straube and Ozsu [117] proceeds in two stages: logical and physical expression optimizations. In the first stage, a logical query expression is translated into an equivalent but more efficient form according to algebraic transformation rules. There are two types of algebraic rewrite rules: pure algebraic rewrite rules and semantic rewrite rules. The former rules can be applied to any expression if there is a pattern match of subexpressions. However, the latter are applicable only when additional conditions on a database schema are met. Hence the

semantic rules are specific to an application. In the second stage, an optimal logical query expression from the first stage is mapped into a sequence of data manipulations at an object manager level. Different mappings may lead to different query evaluation plans. Object storage and statistical information can be used to determine an optimal plan for a query. Blakeley, McKenna and Graefe [16] describe the Open OODB query optimizer generated from Volcano Optimizer Generator [42]. Query optimization is performed in two stages as well. An input query to the optimizer is expressed in an algebraic form similar to a relational one. The query is first simplified into an equivalent but more efficient form according to the transformation rules, and logical and physical properties. Then the correspondence between algebraic operators in a query and execution algorithms is established with implementation rules. This process generates an optimal evaluation plan.

Guo et al. [44] present a unique algebra called association algebra. The domain of the algebra operations is a set of associated patterns which are collections of objects connected via attribute relationships and non-associations. The algebra maintains closure property, i.e., both input and output of an algebraic operation are associated patterns. It is not necessary to introduce equality or identity as in [110, 101] since the algebraic operations directly manipulate associated patterns which preserve structural information of objects as well as object identities. Even though the algebra is quite unique, the query optimization technique is quite similar to the other algebraic transformation-based approaches.

3.5 Problems in DOOD Query Evaluation

This section identifies several important open or not well-solved problems in DOOD query evaluation and reports research progress towards the problems. The problems include navigation through complex objects, query optimization in the presence of complex selections, joins and aggregations, user-defined methods and encapsulation, and recursive query evaluation.

3.5.1 Navigation through Complex Objects

DOOD supports complex data objects and enables explicit and natural representation of logical relationships among complex objects via class/subclass hierarchies, attributes, methods, object identities, etc. Thus, navigation among different classes and objects via class composition hierarchies and/or class hierarchies is an essential operation. A navigation from one object in a class to objects in other classes is essentially a "pointer chasing" (using object identity "OID" references) operation which may cause significant performance degradation because the objects to be accessed may be stored at widely scattered locations and many disk read operations may be required to fetch them into main memory [39]. The attempts to solve this problem can be classified into three classes of techniques: indexing methods which includes nested indices [94, 15, 14, 28, 24] for associative search and navigation index structures [68, 57, 130], replication and caching [111, 65], complex object assembly [66], and read-ahead buffering [102, 31].

In Maier and Stein [94], a series of index components, indices on each level of the nested attributes, are maintained for the purpose of update propagations. In Bertino and Kim [15], three index structures are presented: the nested index, path index and multiindex, which have been later extended to handle inheritance of classes appearing in a path expression [14]. The nested index structure facilitates associative search and update by storing together the key values of the tail attribute, the objects of the head class, and the intermediate objects in a path expression into primary records. An auxiliary index, which basically keeps the direct reference information between objects, together with the information in primary records is used to propagate updates. The nested index structure in general outperforms the other two index structures [14]. Choenni et al. [28] propose an optimal index configuration by splitting a long path expression into some shorter ones, and by indexing the shorter paths with the index structures in [15, 14]. Chawathe, Chen and Yu [24] take index interaction into consideration when selecting a set of nested indices for nested object hierarchies. The index interaction refers to the phenomenon that the inclusion of one index might have impact on the benefit obtained by the other indices if the former is overlapped

with the latter ones. The problem of selecting an optimal index scheme is formulated as an optimization problem against an objective function. The experiment shows the index selection does improve the overall performance. These approaches only support associative retrieval of objects through nested attributes but not navigations in both directions along a reference chain.

Kemper and Moerkotte [68] present a data structure, called the access support relation, which keeps the identifiers of the objects connected by the attribute relationships in a path expression and can span over the reference chains of a path expression. Several alternatives which include the full, canonical, left and right extensions and the decomposition of access support relations for a given path expression are discussed. The optimal one is determined according to the domain-specific information such as the probabilities of different types of queries and updates. The storage size of each component in an access support relation could be large because all the identifier sequences of the joinable objects along an object path corresponding to the component are stored, and any two objects in two classes could be connected by more than one object path. Further, an update on one object may need to be propagated to several components or to the entire access support relation, which could be costly. Hua and Tripathy [57] propose a navigation structure, called the object skeleton, which essentially is a network of object identifiers. Two object identifiers are connected if the corresponding objects are associated by, for example, an attribute relationship. The approach is more general in the sense that the navigations can be supported between two classes not only in a path expression but also over a network of classes. The navigations, however, are supported efficiently only if the starting points of the navigations can be located by using some nested indices such as those in [15, 14]. Besides, an update is required to be propagated over the network of object identifiers and the nested indices.

Shekita and Carey [111] describe a mechanism, called field replication, which replicates the values of nested attributes. In-place field replication stores the replicated data with the objects, whereas separate field replication stores the replicated data in a separated place. The separated replication is used to solve the issue of updating the shared replicated data. Inverted path structures, which are similar to the index

components in [94], are used to support update propagation. Kato and Masuda [65] present a mechanism, called persistent caching, which is similar to the field replication [111]. In this approach, the referenced objects are cached into the referencing objects. Update is delayed until the cached objects are required. A hash table stored in the main memory is employed to maintain the cached values consistent with the original objects. These approaches support only forward navigations along a reference chain. Besides, extra mechanism and information are needed in order to maintain the replicated or cached data consistent with the original data.

Keller, Graefe and Maier [66] propose an assembly operator which efficiently translates a set of complex objects from their disk representations to quickly traversable memory representations. The complex objects could be one or more objects connected by "inter-object" references. Templates are used to store structural and statistical information of objects. Component iterators use the templates to determine what parts of a complex object to assemble, when assembly is complete or how to find unresolved references within a newly retrieved object. The assembly operator uses the templates and component iterators to selectively and intelligently assemble complex objects.

Palmer and Zdonik [102] propose saving past reference patterns for predicting future object faults. If accesses similar to the stored patterns are detected, read-ahead is activated to prefetch the likely required pages. Curewitz, Krishnan and Vitter [31] discovered the similarity between data compression and prefetching. The intuition is that a data compressor usually compresses data by assuming a dynamic probability distribution on the data. A data compressor encodes highly expected data with fewer bits but unexpected data with more bits. Therefore, if a data compressor successfully compresses data, its assumption of probability distribution on the data must be realistic and can be employed effectively for predicting and prefetching data. Three data compression algorithms are applied to prefetching. The experiment results show that the prefetches based on the data compression methods achieve more significantly reduced object fault rates than a pure LRU. If references fit well into the expected pattern, the prefetching is effective. On the other hand, any prefetching methods need to do some "guess work". The performance could be suffering if some changes occur in the reference pattern.

3.5.2 Queries Including Complex Selections, Joins and Aggregations

Navigation operations are frequently expressed in the form of path expressions, a main syntactic notion in DOOD languages. Although navigations are performed over a collection of objects which are connected by some logical relationships and are stored in different pages or blocks on disks, the navigations are often constrained to some selective objects. These constraints are represented by selection and join conditions, sometimes with aggregation functions. It is, therefore, important to investigate how these constraints, i.e., selections and joins, can be employed to efficiently process these navigations.

Kemper and Moerkotte [69] propose a rule-based query optimizer which uses access support relations to evaluate path expressions. Several rules are presented to handle path expressions, e.g., prolonging path expressions, splitting path expressions, etc.

Cluet and Delobel [29] transform pointer chasing operations into join operations. By introducing typing and intermediary variables in path expression-based algebraic expressions, they make it possible to employ equivalences which can not be otherwise employed. The formalism also allows factorization of common sub path expressions in a query.

In Blakeley, McKenna and Graefe [16], a logical algebra operator, called materialize or Mat, is proposed to optimize the evaluation of path expressions. It explicitly indicates the use of the inter-object references in a path expression. A materialize operator can be transformed into joins if the"scope" introduced by a materialize operator is a"scannable" object. The joins can be implemented by join methods such as the hybrid hash join and the assembly operator.

These approaches address the matter of efficiently evaluating path expressions by using access support relations or translating them into joins. However, the issue of whether and how the constraints on path expressions can be employed effectively to evaluate the path expressions was not investigated in these studies. Chapter 5 proposes "Push constraint condition inside navigation" to deal with the issue.

3.5.3 Methods and Encapsulation

Query evaluation in the presence of encapsulated methods is a challenging issue. In most of the previous studies, e.g., [8, 90, 29, 110, 101, 117], encapsulated methods are considered as black boxes. Consequently some optimal query evaluation plans may be excluded from consideration.

Encapsulation is good for effective software maintenance. However, encapsulation hides the implementations of methods and blocks the information required for query optimization. Graefe and Maier [41] and Daniel et al. [32] propose the revelation of encapsulated methods to query optimizers. Their papers indicate that methods may eventually be transformed into algebraic expressions and substituted by the corresponding algebraic expressions. Thus queries can be optimized by a conventional query optimizer, e.g., the EXODUS extensible query optimizer. However, the transformations of arbitrary methods into algebraic expressions are not presented in their papers.

Vandenberg and DeWitt [124] take into account the optimization of queries including methods. Methods are coded by the EXCESS query language and therefore, can be substituted by the query language codes during the optimization process. In both Beeri and Kornatzky [11] and Jiao and Gray [64], the revelation of encapsulated methods is discussed. However, both queries and methods are written in the same functional languages. Definitions of methods can be fully revealed to query optimizers. In these approaches, only methods coded in query languages are taken into consideration during query optimization. Thus the application of methods is restricted.

Methods are usually defined by users in an arbitrary language, and in many cases they are coded in a procedural language such as C++. A query optimizer may perform more effective optimization if it can "understand" the user-defined methods. Chaudhuri and Shim [23] consider the query optimization in the presence of user-defined functions. The semantic information about these foreign functions is expressed in a declarative rule language. Queries could be transformed into equivalent but more efficient forms according to transformation rules. The traditional relational cost model

is extended to accommodate the presence of foreign functions. The traditional join reordering algorithm based on dynamic programming is modified to search for optimal query evaluation plans.

In traditional database query optimization, it is often assumed that selections are simple and inexpensive. It is, therefore, preferable to perform selections before joins in order to reduce the sizes of join relations. However, this assumption is not valid any more if selections include expensive user-defined methods. Hellerstein and Stonebraker [56] consider the issue of optimizing queries with expensive predicates. The problem is formulated as placing expensive predicates in an optimal join plan such that the total cost including the cost of joins and selections is minimized. An algorithm, called predicate migration, is implemented in POSTGRES. The experiment shows that the performance gain could be significant. Hellerstein [55] continues this work and presents a family of algorithms: PushDown, PullRank, modified Predicate Migration and PullAll. The implementation in Montage and performance study show that predicate migration provides good query evaluation plans over a wide range of queries with expensive predicates.

Semantic and cost information of user-defined methods is useful for efficient query evaluation, however, neither of them is easy to obtain in practice. In addition, the incompatibility between declaratively defined query and procedurally coded users' methods, and mismatch between set-oriented evaluation of queries and object-at-a-time computation of methods are not well solved. Precomputing or materializing user-defined methods represents an alternative approach. Instead of computing methods in an object-at-a-time fashion at run time, access of the materialized results is performed in a set-oriented way. Materialization is especially effective and beneficial when methods are expensive to compute. However, the crucial issue for materialization is update maintenance. Kemper, Kilger and Moerkotte [67] describe several tuning strategies for method materialization maintenance. The main goal is to minimize the overhead of invalidation and rematerialization upon update operations. Objects involved in materialization are distinguished from non-involved objects. The information of which attributes of those involved objects are accessed during materialization can further be exploited to decreased the overhead. Some operational semantics about methods

can also be employed to reduced maintenance costs. For example, the transformation *scale* is the only one that may invalidate the precomputed values of *volume* while *rotate* and *translate* do not.

3.5.4 Recursive Query Evaluation

Although recursive query evaluation has been explored in the context of deductive database systems, little has been investigated on how DOOD features impact the recursive query evaluation and whether the recursive query evaluation methods developed in deductive database systems can be extended to handle DOOD recursive queries.

There are research prototypes which integrate object-orientation with deductive database languages. However, these systems typically extend deductive database languages such as datalog with limited object-oriented features [43, 63] or a C++ type system [114]. Programs are first translated into Horn clause-like programs such as datalog, and then evaluated with the existing deductive query evaluation methods.

COMPLEX [43] integrates datalog with limited object-oriented features such as object identity, complex objects and inheritance. The COMPLEX program can be translated into datalog program by incorporating rules which enforce object-oriented features. The deductive query evaluation method, query and subquery [126, 127], is employed to evaluate recursive queries.

Similarly, ConceptBase [63] incorporates structural aspects of object-orientation into a deductive database language. The object-oriented features such as object identity, class and inheritance are represented as integrity constraints. Therefore, the object deductive database system can be viewed as a deductive database system with integrity constraints. Semantic query optimization is employed to optimize queries.

Coral++ [114] extends Coral [103], a deductive database system, with a C++ type system. Class definitions are handled by a C++ compiler. The system accommodates accessing named attributes and invoking methods by generating predicates which perform appropriate attribute accesses and method invocations. Therefore, method name binding and invocation are left to a C++ compiler. The transformed Coral++

programs can be evaluated by using the existing Coral system.

Although object-orientation is not explicitly supported, XSB [107] provides a platform for implementing object-orientation with Hilog [25] syntax. XSB implements Hilog by transforming higher-order terms into first-order forms with *apply/N*, and then compiling and optimizing the first-order forms.

Chapter 4

Join Index Hierarchies for Efficient Navigations

4.1 Introduction

Navigation is an essential operation for exploring logical relationships among complex objects via class/subclass hierarchies, attributes, methods, object identities, etc. For example, in Figure 4.1, several classes are connected via the relationships induced by attributes, methods and class/subclasses. To find out which departments offer the courses taken by the assistant professor Jones' students, navigation is performed from the object "jones" in the class *AssistantProf* to the objects in the class *DEPT* via the subclass/class relationship of *AssistantProf* and *PROF*, the method *Supervise* of *PROF*, the attribute *TakeCourses* of *STUDENT*, and the attribute *Dept* of *COURSE*. The objects in the intermediate classes *STUDENT* and *COURSE* have to be accessed in order to find out the objects in the class *Dept* which have the logical relationship, expressed in the query, with the object "jones" in the class *AssistantProf*. Navigations may jeopardize the system performance because the objects along a navigation path may reside at widely scattered locations and many disk read operations may be required to fetch them into main memory.

Following the philosophy of indexing methods, a join index hierarchy method is proposed in this thesis, which extends the join index technique developed in relational

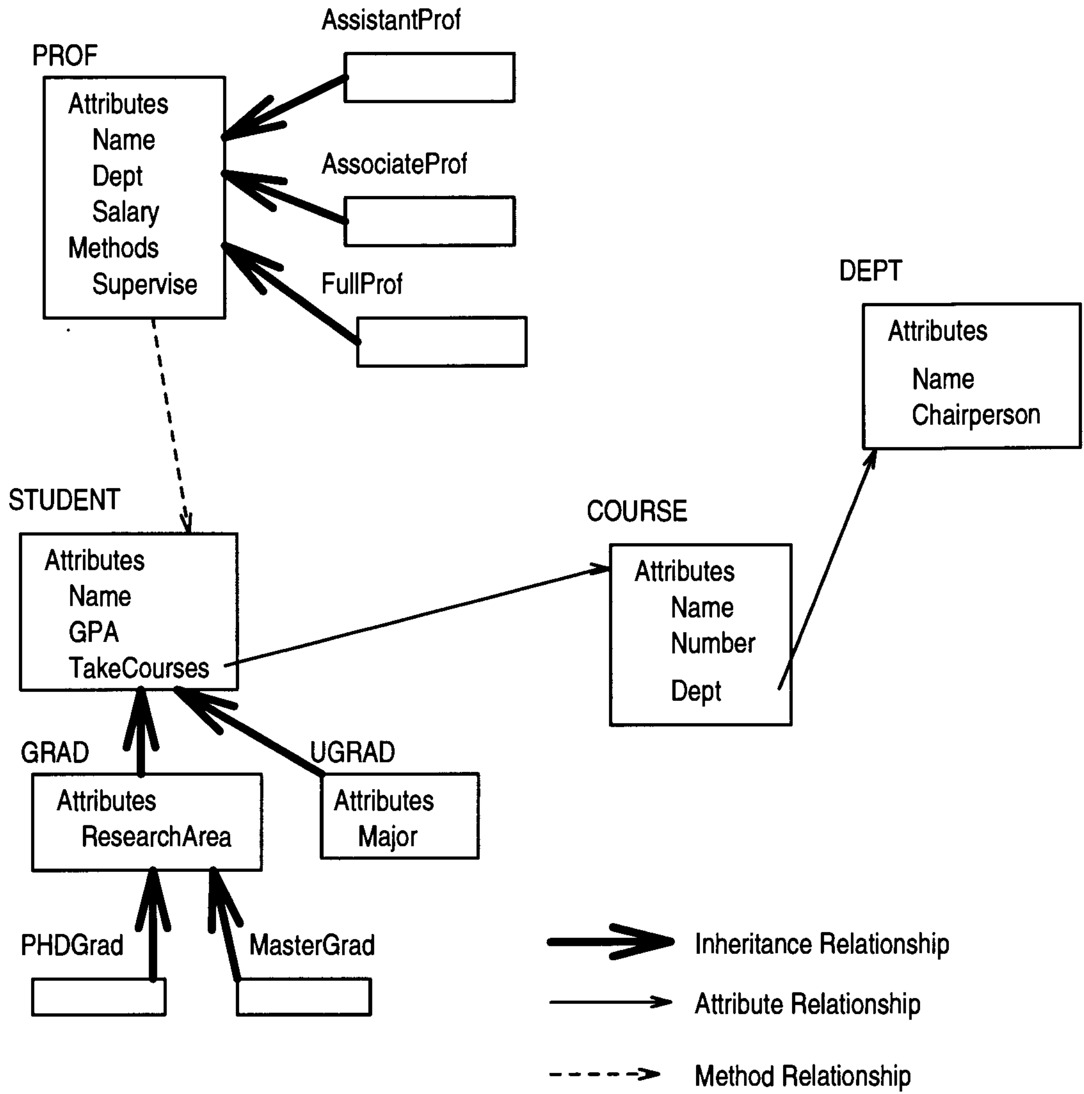


Figure 4.1: Navigation.

databases [123] and its variations in spatial databases [106, 92], constructs hierarchies of join indices to accelerate navigations via a sequence of objects and classes. In a broad sense, a join index here stores the pairs of identifiers of objects of two classes that are connected via direct or indirect logical relationships. Those formed by direct logical relationships are called base join indices; whereas those representing indirect logical relationships are called derived join indices. Base and derived join indices form a join index hierarchy. A join index hierarchy supports navigations through a sequence of classes in either a forward or a backward navigation direction and supports efficient update propagation starting with the base join indices by localizing update propagations in the hierarchy.

The following considerations motivate the proposal of the join index hierarchy structures.

- First, by construction of join index hierarchies, the "pointer chasing" problem, that is, accessing objects and their properties via a sequence of referencing pointers to widely scattered disk locations, is transformed into simple accessing of appropriate join index files. This may significantly reduce the I/O accessing cost in object-oriented query processing. The price for this I/O cost reduction is the increase of space for storing join index files, which is practically implementable since large inexpensive disk memories are available with reasonable cost based on the current hardware technology.

- Secondly, with join index hierarchies, appropriate join index files for specific navigation operations can be selected by consulting the index hierarchy directory. Moreover, update propagation can be localized to a few base and derived join index files in the hierarchy. Both forward and backward navigations can be supported with minimum storage and update overheads.

- Thirdly, using join index hierarchies, object-at-a-time navigation is transformed into efficient, set-oriented and associative access of join indices. Moreover, it supports navigations among objects connected not only via a sequence of attribute relationships but also via a sequence of methods and deduction rules.

This is accomplished by precomputing methods and rules and storing the related information in join indices. By doing so, the object-at-a-time evaluation of computationally intensive methods or deduction-intensive rules can be transformed into efficient and set-oriented accessing of precomputed relationships. Moreover, retrieval from either direction becomes available even for methods and deduction rules.

- Fourthly, in some cases, the join of some classes on certain attributes may generate a substantially large join index file because of its large join selectivity, or some class may sustain regular and frequent updates. Joins involving such kind of characteristics should be considered as "fire walls" in the construction of join index hierarchies. The system should prohibit the construction of such join indices or the merge of such join indices into the hierarchy in order to avoid the potential explosion on the size of join index files or the heavy cost of updates. Queries involving such joins can be processed by performing concrete joins or using the base join index files, if available.

4.2 Preliminaries

A join index hierarchy structure is proposed here to support efficient navigation through multiple object classes. For example, in Figure 4.1, one may like to find which departments offer courses taken by Jones' students, or which courses the undergraduate student "John" is taking, which departments offer the courses taken by a PhD student "Mary", etc. These queries correspond to navigations through a set of classes, such as *AssistantProf*, *DEPT*, *UGRAD*, *COURSE*, etc. via appropriate relationships.

The variations of a join index hierarchy can be constructed based on the richness of the derived join index structures. Three kinds of structures: *based-only*, *complete*, and *partial*, are investigated in terms of their construction, navigation and update propagation.

For the clarity of presentation, only the relationships between the existing attributes among object classes are considered in the construction and maintenance of

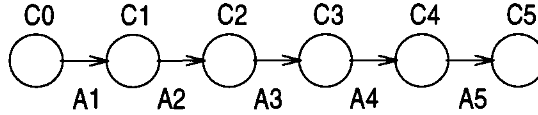


Figure 4.2: A Schema Path of Length 5.

join index hierarchies. A join index hierarchy which handles the relationships induced by attributes, methods, rules, and class/subclass hierarchies will be discussed later.

A database schema is a directed graph in which the nodes correspond to classes, and edges to relationships between classes. Suppose A_k is an attribute of class C_i, and A_k ranges over class C_j. Then there exists a directed edge from C_i to C_j in the schema graph, labeled with A_k. Moreover, if for $i = 0, 1, \ldots, n-1$, there is a directed edge from C_i to C_{i+1}, labeled with A_{i+1}, in a database schema, then $\langle C_0, A_1, C_1, A_2, \ldots, A_n, C_n \rangle$ is a schema path.

Regarding to a schema path $\langle C_0, A_1, C_1, A_2, \ldots, A_n, C_n \rangle$ over a database schema, a join index file (node) $JI(i,j)$ $(1 \leq i < j \leq n)$ consists of a set of tuples $(OID(o_i), OID(o_j), m)$, where o_i and o_j are objects of classes C_i and C_j respectively, and there exists an object path $\langle o_i, o_{i+1}, \ldots, o_{j-1}, o_j \rangle$ such that for $k = 0, 1, \ldots, j-i-1$, o_{i+k+1} is referenced by o_{i+k} via the attribute A_{i+k+1}, and m is the number of the above distinct object paths that connect the objects o_i and o_j.

Definition 4.1 Join index hierarchy. Join index nodes connecting different object classes along a schema path form a join index hierarchy, denoted as $JIH(C_0, A_1, C_1, A_1, \ldots, A_n, C_n)$, or simply $JIH(0, n)$. The longest join index path, $JI(0, n)$, is the root of the hierarchy. Each node $JI(i, j)$ where $j - i > 1$ may have two direct children $JI(i, j-k)$ and $JI(i+l, j)$ where $0 < k < j - i$ and $0 < l < j - i$. The join index nodes $JI(i, i+1)$, for $i = 0, 1, \ldots, n-1$, are at the bottom of the hierarchy, and are therefore, called base join indices.

Figure 4.2 shows a schema path of length 5 on a class composition hierarchy and Figure 4.3(a)(b)(c) illustrates the following three join index hierarchy structures.

1. A **complete join index hierarchy (C-JIH)**, as shown in Figure 4.3(a), consists of a complete set of all the possible base and derived join indices. It

supports navigations between any two directly or indirectly connected object classes along the schema path.

2. A **base join index "hierarchy" (B-JIH)**, as shown in Figure 4.3(b), consists of only base join indices. It supports direct navigations only between any two adjacent classes. It cannot be entitled as a "hierarchy" in a rigorous sense but can be viewed as a degenerate hierarchy with all the higher level join index nodes missing, where the missing nodes can be derived from the base join indices.

3. A **partial join index hierarchy (P-JIH)**, as shown in Figure 4.3(c), consists of a proper subset of the set of base and derived join indices in a complete join index hierarchy. It supports direct navigations between a pre-specified set of object class pairs since it materializes only the corresponding join indices and their related auxiliary (derived) join indices.

Figure 4.3(c) demonstrates a typical partial join index hierarchy which supports direct navigations between C_0 and C_4, and C_2 and C_5. Their corresponding JI nodes: $JI(0,4)$ and $JI(2,5)$, circled in the figure, are called target nodes. Notice that a materialized intermediate level node $JI(i,j)$ may be used not only for supporting navigations between C_i and C_j but also (and sometimes more importantly) for accelerating update propagations from the base join indices to higher level join indices such as $JI(0,4)$.

For example, if there were no intermediate level join index nodes in the hierarchy $JIH(0,5)$, four join-like (defined later) operations are needed on average to propagate an update from the base join indices to the target nodes $JI(0,4)$ and $JI(2,5)$. With the help of intermediate level join indices, it takes an average of 2.2 join-like operations to propagate an update from the base join indices.

In a join index hierarchy $JIH(0,n)$, the base join index nodes $JI(i,i+1)$ (for $i=0,\ldots,n-1$) reside at level 1, and the root node $JI(0,n)$ at level n. Although a complete join index hierarchy could be quite large, each individual join index node is usually of reasonable size. In many cases, it is unnecessary to materialize all of the join index nodes in the hierarchy since it is beneficial to support only the frequently

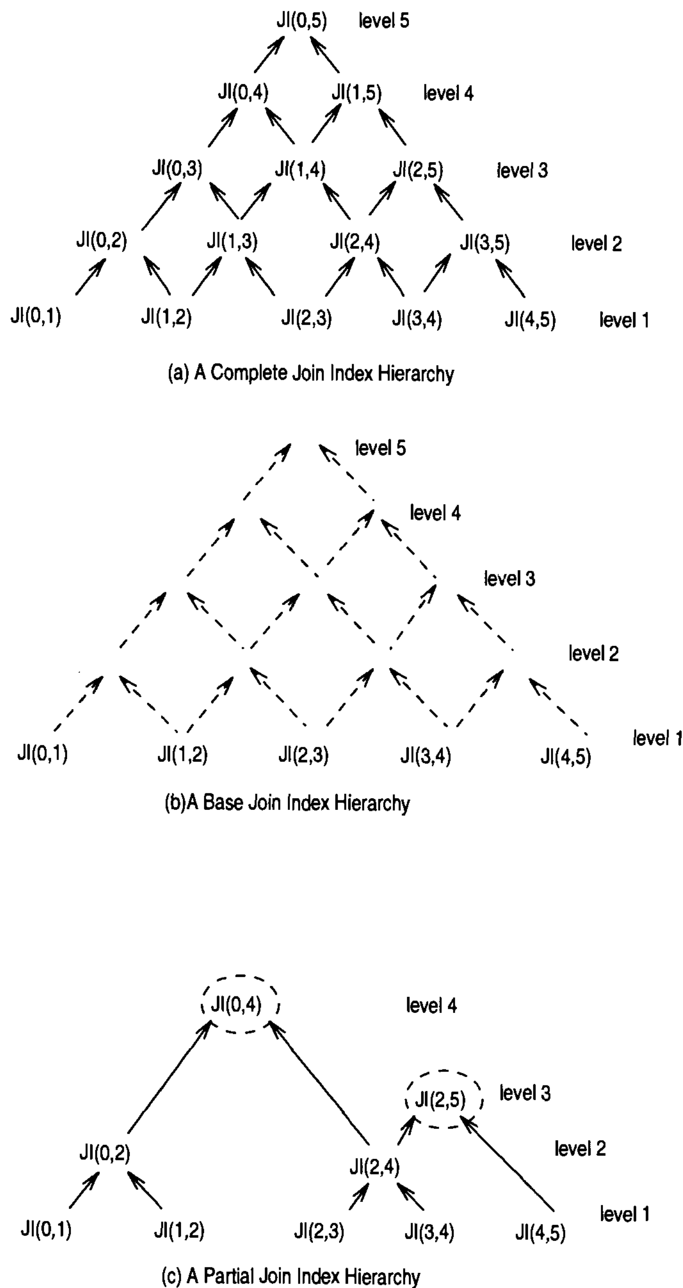


Figure 4.3: Three Kinds of Join Index Hierarchies Corresponding to the Schema Path in Figure 4.2

used navigations. Given a set of frequently accessed schema paths, a partial join index hierarchy can be constructed to support the corresponding navigations.

Definition 4.2 Target and auxiliary join index. In a join index hierarchy, a set of join index nodes which must be supported (due to frequent references) are called target join indices; whereas the others which are mainly used for update propagation are called auxiliary join indices.

For example, $JI(0,4)$ and $JI(2,5)$ in Figure 4.3(c) are target join indices while $JI(0,2)$, and $JI(2,4)$ in Figure 4.3(c) are auxiliary join indices. Auxiliary join indices can of course be used, as a by-product, for support of the navigations between the corresponding classes. The target, auxiliary and base join indices are materialized join indices. The unmaterialized join indices are called virtual join indices.

Update propagation includes three types of updates.

1. *Insert*[1] an attribute relationship A_{i+1} between an object o_i in class C_i and an object o_{i+1} in class C_{i+1}. This corresponds to inserting a tuple ($OID(o_i)$, $OID(o_{i+1})$, 1) to the base join index $JI(i, i+1)$.
2. *Delete*[2] an attribute relationship A_{i+1} between an object o_i in class C_i and an object o_{i+1} in class C_{i+1}. This corresponds to deleting a tuple ($OID(o_i)$, $OID(o_{i+1})$, 1) from $JI(i, i+1)$;
3. *Modify* an attribute relationship A_{i+1} from that between an object $o_i \in C_i$ and another object $o_{i+1} \in C_{i+1}$ to that between $o_i \in C_i$ and $o'_{i+1} \in C_{i+1}$. This corresponds to deleting an existing tuple $(OID(o_i), OID(o_{i+1}), 1)$ from $JI(i, i+1)$ and inserting a new tuple $(OID(o_i), OID(o'_{i+1}), 1)$ to $JI(i, i+1)$.

As a notational convention, $\triangle JI(i,j)$ denotes a set of tuples being inserted into $JI(i,j)$. $\triangle JI(i,j)$ consists of tuples $(OID(o_i), OID(o_j), m)$ and $m > 0$, indicating that there are m new object paths connecting o_i and o_j. Similarly, $\bigtriangledown JI(i,j)$ represents a set of tuples being deleted from $JI(i,j)$. It consists of tuples $(OID(o_i), OID(o_j), -m)$ and $m > 0$, indicating that there are m object paths connecting o_i and o_j being

[1] Inserting an existed relationship between two objects is ignored.

[2] Deleting a non-existed relationship between two objects is ignored.

deleted. $\delta JI(i,j)$ denotes $\bigtriangledown JI(i,j)$ followed by $\triangle JI(i,j)$. A join operator "$\bowtie_c$", which is similar to a join operation in relational databases, and another operator "$\bigcup_c$", used in the update algorithm, are introduced as follows.

Definition 4.3 $\bowtie_c$. $JI(i,k) \bowtie_c JI(k,j)$ contains a tuple $(OID(o_i), OID(o_j), m_1 \times m_2)$ if there is a tuple $(OID(o_i)$, $OID(o_k)$, $m_1)$ in $JI(i,k)$ and a tuple $(OID(o_k)$, $OID(o_j)$, $m_2)$ in $JI(k,j)$. That is, if there are m_1 distinct object paths from o_i to o_k and m_2 distinct object paths from o_k to o_j, there are $m_1 \times m_2$ object paths from o_i to o_j. Notice that identical tuples, such as $(OID(o_i), OID(o_j), m_k)$ (for $k = 0, 1, \ldots, p$) are automatically merged into one with their path numbers accumulated, i.e., $(OID(o_i), OID(o_j), \sum_{k=0}^{p} m_k)$.

Definition 4.4 $\bigcup_c$. $JI(i,j) \bigcup_c \triangle JI(i,j)$ indicates an insertion into $JI(i,j)$. If there exists a path in $JI(i,j)$ for the corresponding objects, the number of paths connecting o_i and o_j will increase. Similarly, $JI(i,j) \bigcup_c \bigtriangledown JI(i,j)$ indicates a deletion from $JI(i,j)$, and the number of paths connecting the corresponding objects o_i and o_j will decrease.

4.3 Construction and Maintenance of Join Index Hierarchies

4.3.1 Construction of a Partial Join Index Hierarchy

A partial join index hierarchy can be constructed in three steps: (1) find a set of necessary auxiliary join indices for a given set of target indices; (2) build the corresponding base join indices; and (3) build the target and auxiliary join indices from the lowest level up.

Example 4.1 In Figure 4.3(a), the join index $JI(1,5)$ can be computed from $JI(1,4)$ and $JI(4,5)$, where $JI(1,4)$ can be derived in turn from $JI(1,3)$ and $JI(3,4)$, and $JI(1,3)$ from $JI(1,2)$ and $JI(2,3)$.

The base join indices for $JI(1,5)$ are the set:

$$\{JI(1,2), JI(2,3), JI(3,4), JI(4,5)\}.$$

The auxiliary join indices for supporting efficient update of $JI(1,5)$ are:

$$\{JI(1,4), JI(1,3)\}.$$

Notice that there could be other choices in selecting auxiliary JIs, such as $\{JI(1,3), JI(3,5)\}$, etc. □

Example 4.2 To directly support the navigations between C_0 and C_4, and C_2 and C_5, the set of target join indices are $\{JI(0,4),\ JI(2,5)\}$, and the set of base join indices are

$$\{JI(0,1), JI(1,2), JI(2,3), JI(3,4), JI(4,5)\}.$$

Three different kinds of partial join index hierarchies are presented in Figure 4.4(a)(b) and Figure 4.5.

The sets of auxiliary JIs which supports the two target JIs are $\{JI(0,3), JI(1,3), JI(2,4)\}$ in Figure 4.4(a), $\{JI(1,4), JI(2,4)\}$ in Figure 4.4(b) and $\{JI(0,2), JI(2,4)\}$ in Figure 4.5. □

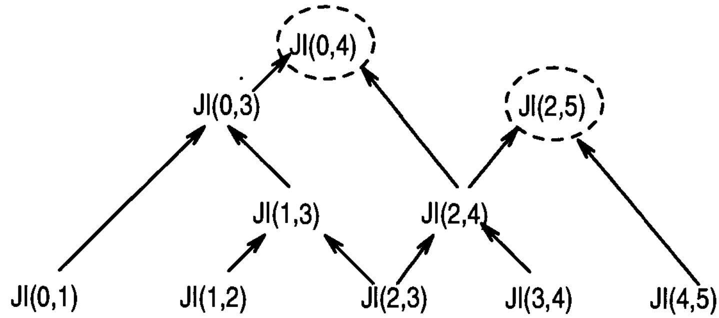
Given a set of target join index nodes, the join index nodes which need to be materialized are the union of the base and auxiliary sets derived from each target join index node. Since there could be more than one choice in the derivation, the optimal choice should be the one which minimizes (1) the total number of auxiliary join indices (and then the total storage costs); and (2) the total number of $\bowtie_c$ operations in updating the target join indices. This is performed by Algorithm 4.1.

Algorithm 4.1 Construction of a minimum auxiliary set of JIs

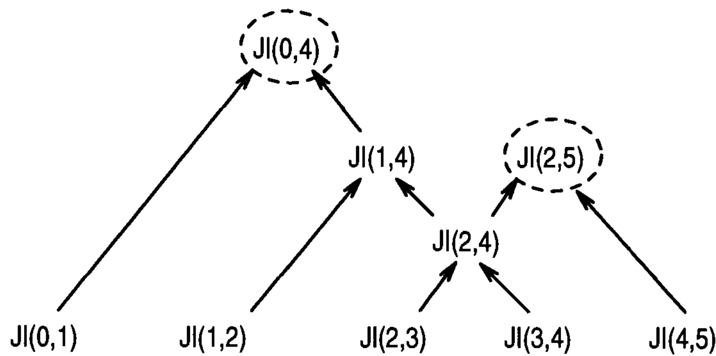
Input: A set of classes $C_0, \ldots, C_n$, and a set of target JI nodes (i.e., frequently referenced class pairs) in the schema path C_0, A_1, C_1, A_2, $\ldots$, A_n, C_n.

Output: A minimum set of auxiliary JIs nodes.

Method: The method collects the set of auxiliary nodes which are used to generate the set of target nodes, and then selects those containing the minimum numbers of nodes, as shown below.



(a) A Partial Join Index Hierarchy Supporting
JI(0,4) and JI(2,5)
avg # of operations for update=2.8



(b)A Partial Join Index Hierarchy Supporting
JI(0,4) and JI(2,5)
avg # of operations for update=2.4

Figure 4.4: Two Partial Join Index Hierarchy Structures for Supporting JI(0,4) and JI(2,5).

1. Starting with the set of target nodes, find S: the set of sets of their immediate auxiliary nodes. Notice that the set of immediate auxiliary nodes for a (target or auxiliary) node $JI(i,j)$ is $\{JI(i,k), JI(k,j)\}$ for $i < k < j$ with the removal of $JI(i,k)$ or $JI(k,j)$ if it is a target node or a base node. If there is an empty set resulted from this removal, return the empty set. Otherwise, if there are more than one such k available, each k generates one set, and the result is a set of sets. Thus, S is in the form of $\{\{JI(i,k), \ldots, JI(k,j)\}, \ldots, \{JI(i,m), \ldots, JI(m,j)\}\}$.

   For each JI in the set s in S, find its immediate auxiliary nodes. If an immediate auxiliary node consists of l sets, $a_1, \ldots, a_l$, make l copies of s, and add each of a_i $(1 < i < l)$ to a copy, which forms l new sets. This process repeats until no new immediate auxiliary nodes can be found. The result is a set of auxiliary node sets which are used for generating the set of target nodes.

2. For each set s in the generated set of auxiliary nodes, count the number of (auxiliary) nodes. Only those with the minimum number of nodes are retained.

3. From the retained sets obtained in Step 2 (i.e., the set in which each set contains the minimum number of auxiliary nodes), calculate the number of $\bowtie_c$ operations required for updating each set and select the one which requires the minimum number of $\bowtie_c$ operations. This is computed by averaging the sum of the numbers of all the $\bowtie_c$ operations needed for propagation of the updates on the base join index nodes. □

Example 4.3 We examine how the algorithm works on Example 4.2. At the beginning,

$$S = \{\{JI(0,4), JI(2,5)\}\}.$$

The target join index $JI(0,4)$ has three immediate auxiliary sets $\{JI(0,3)\}$, $\{JI(1,4)\}$ and $\{JI(0,2), JI(2,4)\}$; whereas the target join index $JI(2,5)$ has two immediate auxiliary sets $\{JI(2,4)\}$ and $\{JI(3,5)\}$. Among these nodes, only $JI(0,3)$ and $JI(1,4)$ have nonempty auxiliary sets. The former has $\{JI(0,2)\}$ and $\{JI(1,3)\}$,

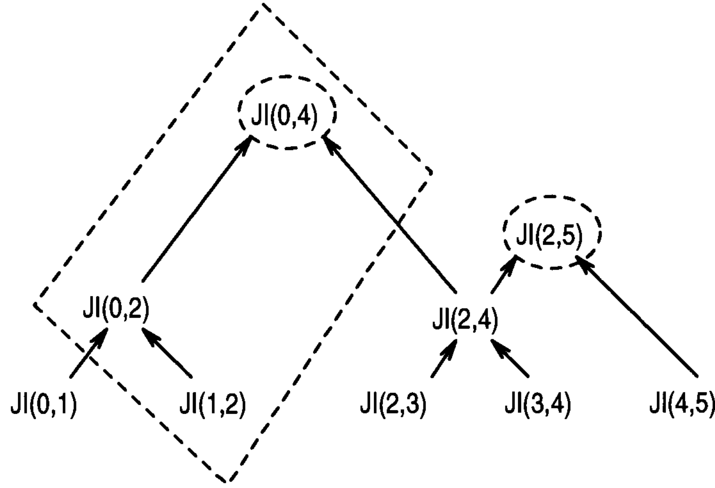


Figure 4.5: Build a Partial Join Index Hierarchy and Propagate Update.

and the latter has $\{JI(1,3)\}$, and $\{JI(2,4)\}$. Therefore, the set of possible auxiliary node sets should be all of their combinations, that is,

$$S = \{\{JI(0,2), JI(0,3), JI(2,4)\}, \{JI(1,3), JI(0,3), JI(2,4)\},$$

$$\{JI(1,3), JI(1,4), JI(2,4)\}, \{JI(1,4), JI(2,4)\},$$

$$\{JI(0,2), JI(0,3), JI(3,5)\}, \{JI(1,3), JI(0,3), JI(3,5)\},$$

$$\{JI(2,4), JI(1,4), JI(3,5)\}, \{JI(0,2), JI(2,4)\},$$

$$\{JI(0,2), JI(2,4), JI(3,5)\}\}.$$

Both $\{JI(1,4), JI(2,4)\}$ and $\{JI(0,2), JI(2,4)\}$ have the minimum number of auxiliary join indices. The first one corresponds to the partial join index hierarchy structure in Figure 4.4(b), whereas the second one to that in Figure 4.5. The average numbers of $\bowtie_c$ operations for update propagation in Figure 4.4(b) and Figure 4.5 are 2.4 and 2.2 respectively. Obviously, the second partial join index hierarchy is the most preferable one. □

Algorithm 4.2 Construction of a partial join index hierarchy.

Input: A set of frequently referenced class pairs (i.e., target JI nodes) in a schema path C_0, A_1, C_1, A_2, ..., A_n, C_n and the corresponding classes.

Output: $JIH(C_0, A_1, C_1, A_1, \ldots, A_n, C_n)$, a partial join index hierarchy which supports navigations between these pairs of classes.

Method: The computation includes both finding the minimum set of auxiliary JI nodes and computing all the necessary JIs.

1. Find the minimum set of auxiliary JIs based on the set of target JIs by using Algorithm 4.1.
2. Build base JIs by computing $JI(i, i+1)$ for $i = 0, 1, \ldots, n-1$ and constructing the corresponding B^+-tree indices on i for each base JI.
3. Build auxiliary and target JIs. This is accomplished by computing the selected auxiliary JIs and/or target JIs from the bottom level up using the $\bowtie_c$ operation, and constructing the corresponding B^+-tree indices on i for each derived JI.
4. Build "reverse" JIs for searching in the reverse direction. (A reverse JI of $JI(i, j)$, $JI(j, i)$, supports the search from class j to class i via the schema path in reverse to that of $JI(i, j)$). $JI(j, i)$ is derived from $JI(i, j)$ by sorting on j in a copy of $JI(i, j)$ and constructing the B^+-tree indices on j. □

Notice that in step 3 there could be more than one pair (but at most $j - i$ pairs) of JIs of lower level nodes which can be used to compute $JI(i, j)$. A cost model should be constructed to determine the minimum cost pair. Moreover, B^+-trees can be used to build JIs for efficient retrieval and for efficient computation of JIs at higher levels.

The join index hierarchy computes the logical relationships between the objects not only in two adjacent classes but also in the "remote" classes linked via a specified schema path. It maintains both forward and backward join indices and supports both forward and backward navigations efficiently.

Furthermore, navigations on the virtual nodes (unmaterialized nodes) can still be performed efficiently using the partial join index hierarchy. For example, any virtual node in Figure 4.5 can be constructed by at most one join of two existing

materialized JI nodes. Actually, it is easy to verify for $n \leq 6$, taking the root of $JIH(0,n)$ as the single target node, there always exists a set of minimum auxiliary nodes, with minimum update cost, and any virtual node in $JIH(0,n)$ can be obtained by at most one join of two existing (base/auxiliary) JI nodes. For example, $\{JI(0,3), JI(3,6), JI(1,3), JI(3,5)\}$ is such a minimum auxiliary node set for $JIH(0,6)$. This implies that any traversal from one object in any class to any other object class along the schema path with length less than 7 will need to search at most two (indexed) JI files using such a small partial join index hierarchy. Since one rarely constructs a $JIH(0,n)$ for $n \geq 7$ in practice, traversal along any subpath of a schema path in both directions can be performed fairly efficiently using the partial join index hierarchy.

4.3.2 Update Maintenance of a Partial Join Index Hierarchy

An update in one class or in the relationship of one class with another may cause the update of a base join index, such as $JI(k, k+1)$ (and its update is denoted as $\delta JI(k, k+1)$). Such an update will not affect other base join indices but may affect some corresponding join indices at higher levels. It is easy to show that for an update on $JI(k, k+1)$, only the materialized $JI(i,j)$ with $i \leq k$ and $j > k$ will need to be updated accordingly. For example, if $JI(1,2)$ is updated in Figure 4.5, only those join indices in the dotted quadrangle need to be updated.

Algorithm 4.3 Update propagation in a join index hierarchy.

Input: A join index hierarchy $JIH(0,n)$ and $\delta JI(k, k+1)$.

Output: An updated join index hierarchy.

Method: Perform a bottom-up incremental update propagation starting at the base join index.

1. Update the base join index $JI(k, k+1)$ based on $\delta JI(k, k+1)$.

2. Update the auxiliary JIs and/or target JIs from the bottom level up using the $\bowtie_c$ operation. This is implemented as follows.

for level $l := 2$ to n do
  for $i := 0$ to $n - l$ do
    if $JI(i, i + l)$ is an auxiliary or target JI and $i \leq k$ and $i + l > k$
    then incrementally update $JI(i, i + l)$ to $JI'(i, i + l)$.
      Note: This is performed as follows.
      $\delta JI(i, i + l) := JI(i, i + p) \bowtie_c \delta JI(i + p, i + l)$, or
      $\delta JI(i, i + l) := \delta JI(i, i + q) \bowtie_c JI(i + q, i + l)$,
      where $1 \leq p < k - i$ and $k - i \leq q \leq l - 1$;
    $JI'(i, i + l) := JI(i, i + l) \ \bigcup_c \ \delta JI(i, i + l)$; □

Notice that incremental updates are performed on both forward and backward join indices. Also, there could be more than one way to compute $\delta JI(i, i + l)$ in Step 2, and the choice can be determined by a cost analysis.

4.3.3 Base and Complete Join Index Hierarchies

A base join index hierarchy (BJIH) can be constructed and updated in a way simpler than Algorithms 4.2 and 4.3 (only Step 1 of the algorithms need to be performed) since BJIH is a degenerate hierarchy and no upward propagation need to be considered.

However, navigation between C_i and C_{i+l} in a base join index hierarchy requires the retrieval of a sequence of l base join indices:

$$JI(i, i + 1), \ldots, JI(i + l - 1, i + l).$$

This is the major overhead of the base join index hierarchy in comparison with the partial join index hierarchy which requires the retrieval of only one or a very small number of join indices.

Since all the join indices are materialized in a complete join index hierarchy (CJIH), Step 1 of Algorithm 4.2 does not need to be performed in the construction of CJIH: All of the join indices at each level are considered as target join indices. The retrieval

Table 4.1: Database Parameters

Parameters	Meaning, Derivation and Default Values
$\lvert C_i \rvert$	number of objects in class C_i
$\lvert\lvert C_i \rvert\rvert$	number of pages or blocks of class C_i
f_i	average number of references from an object in C_i to objects in C_{i+1} (fan-out)
r_i	average number of objects in class C_i referencing the same object in C_{i+1} $(=\frac{\lvert C_i \rvert * f_i}{\lvert C_{i+1} \rvert})$
$sz(OID)$	number of bytes for storing an object identifier $(= 8)$
$sz(m)$	number of bytes for the counter in a tuple of a join index $(= 4)$
$sz(ji)$	number of bytes of a tuple in a join index $(= 2 * sz(OID) + sz(m))$
$sz(p)$	number of bytes of a page pointer $(= 4)$
B	number of bytes in a block or page of a disk $(= 4096)$
α	average page occupancy factor$(= 70\%)$
BT_f	fan out of a B^+-tree $(\lceil = \frac{\alpha * B}{sz(p)+sz(OID)} \rceil)$
$fwd(i,j,k)$	average number of distinct objects in C_j referenced by a set of k objects in C_i
$bwd(i,j,k)$	average number of distinct objects in C_i referencing a set of k objects in C_j
$\lvert JI(i,j) \rvert$	number of tuples in $JI(i,j)$
$\lvert\lvert JI(i,j) \rvert\rvert$	number of blocks or pages of $JI(i,j)$

could be faster using a complete JIH in comparison with that using a corresponding partial JIH if the retrieval requires to access a (virtual) node which is not directly materialized in the partial JIH. However, a complete JIH obviously takes more storage space and more update propagation cost than a partial JIH although the update algorithm is similar to Algorithm 4.3.

4.4 Performance Evaluation of Join Index Hierarchies

An analytical model is constructed to study the performance of different join index hierarchies, the access support relation [68], a competitive index structure for navigation through a sequence of object classes, and the nested index [15, 14] for associative search. The study is focused on several crucial performance measurements, including

Table 4.2: Database Parameter Values

Parameters	C_0	C_1	C_2	C_3	C_4	C_5
$\|C_i\|$	1000	2000	1000	3000	2000	1000
f_i	1.0s	2.0s	1.0s	1.0s	1.0s	3.0s
s	$s = 0.1, 0.5, 1, 1.5, 2.0, 2.5, 3$					

the storage size of a join index hierarchy, the cost of navigation (query processing), and the cost of update propagation over a join index hierarchy. Table 4.1 lists some database parameters used in the cost analysis. The details of the estimation of some of these parameters are in Appendix A.

4.4.1 Storage and Navigation Costs

The number of pages for a join index $JI(i,j)$ is

$$||JI(i,j)|| = \lceil \frac{sz(ji) * |JI(i,j)|}{B * \alpha} \rceil.$$

Following Valduriez [123], the number of disk accesses for a forward navigation from a set of n_i objects in C_i to objects in C_j using a target join index is

$$1 + y(n_i, \lceil \frac{|C_i|}{BT_f} \rceil, |C_i|) + y(n_i, ||JI(i,j)||, |C_i|),$$

where y is a function from Yao [133],

$$y(k, m, n) = \lceil m * (1 - \prod_{i=1}^{k} \frac{n - \frac{n}{m} - i + 1}{n - i + 1}) \rceil.$$

It represents the number of page accesses for retrieving k objects out of n objects distributed over m pages. Here it is assumed that a typical B^+-tree is of two levels [3]. One page access is needed to retrieve the root node. To find the page pointers for n_i object identifiers, $y(n_i, \lceil \frac{|C_i|}{BT_f} \rceil, |C_i|)$ leaf pages of the B^+-tree are accessed. There are

[3] The results for a B^+-tree of more than two levels can be calculated similarly as in Valduriez [123].

$y(n_i, ||JI(i,j)||, |C_i|)$ pages that need to be accessed to find the tuples corresponding to n_i object identifiers. Thus the number of disk accesses for a forward navigation from a set of n_i objects in C_i to objects in C_j using a base join index hierarchy structure is

$$\begin{aligned}&(1 + y(n_i, \lceil\frac{|C_i|}{BT_f}\rceil, |C_i|) + y(n_i, ||JI(i, i+1)||, |C_i|))\\ &+ \sum_{k=i+1}^{j-1} (1 + y(fwd(i,k,n_i), \lceil\frac{|C_k|}{BT_f}\rceil, |C_k|)\\ &+ y(fwd(i,k,n_i), ||JI(k,k+1)||, |C_k|)).\end{aligned}$$

The first sum is the number of page accesses when the join index $JI(i, i+1)$ is scanned and related tuples retrieved. The second sum covers the case when $fwd(i, k, n_i)$ object identifiers from the previous join index $JI(k-1, k)$ are used to search the join index $JI(k, k+1)$.

4.4.2 Update Cost

Assume that there is an update on an object in C_k which causes the update on $JI(k, k+1)$, either deletion or insertion $\delta JI(k, k+1)$. The cost of updating a partial join index hierarchy consists of three parts. The first part is the cost of updating $JI(k, k+1)$ itself in Step 1 of Algorithm 4.3. The cost of updating forward $JI(k, k+1)$ is

$$\begin{aligned}&1 + y(|\delta JI(k, k+1)|_k, \lceil\frac{|C_k|}{BT_f}\rceil, |C_k|)\\ &+2 * y(|\delta JI(k, k+1)|_k, ||JI(k, k+1)||, |C_k|),\end{aligned}$$

where $|\delta JI(i,j)|_i$ stands for the number of identifiers of distinct objects of C_i in the tuples of $\delta JI(i,j)$ and $|\delta JI(i,j)|_j$ stands for the number of identifiers of distinct objects of C_j in the tuples of $\delta JI(i,j)$. Here $|\delta JI(k, k+1)|_k$ and $|\delta JI(k, k+1)|_{k+1}$ are initialized, e.g., to 1 at the beginning. One page access is needed to retrieve the root node of the B^+-tree of $JI(k, k+1)$. The second sum covers the cost of retrieving the leaf pages of the B^+ tree for finding the page pointers. The third sum handles the cost of inserting or deleting the related tuples which includes reading and writing back the related pages. The cost of updating backward $JI(k, k+1)$ is similar. The

second part is the cost $UJI(i, i+l)$ for updating materialized join index $JI(i, i+l)$ at level l. According to step 2 of Algorithm 4.3, $|\delta JI(i, i+l)|_i$ and $|\delta JI(i, i+l)|_{i+l}$ can be calculated iteratively from $|\delta JI(k, k+1)|_k$ and $|\delta JI(k, k+1)|_{k+1}$. If the first expression in the step 2 of Algorithm 4.3 is chosen, then

$$|\delta JI(i, i+l)|_i = bwd(i, i+p, |\delta JI(i+p, j+l)|_{i+p}), \text{ and}$$
$$|\delta JI(i, i+l)|_{i+l} = |\delta JI(i+p, j+l)|_{j+l}.$$

If the second expression in the step 2 of Algorithm 4.3 is chosen, then

$$|\delta JI(i, i+l)|_i = |\delta JI(i, k+q)|_i, \text{ and}$$
$$|\delta JI(i, i+l)|_{i+l} = fwd(i+q, i+l, |\delta JI(i, i+q)|_{i+q}),$$

where $1 \leq p < k-i$ and $k-i \leq q \leq l-1$. Also, if the first expression in the step 2 of Algorithm 4.3 is chosen, $UJI(i, i+l)$ is calculated as

$$1 + y(|\delta JI(i+p, i+l)|_{i+p}, \lceil \frac{|C_{i+p}|}{BT_f} \rceil, |C_{i+p}|)$$
$$+y(|\delta JI(i+p, i+l)|_{i+p}, ||JI(i, i+p)||, |C_{i+p}|).$$

If the second expression in step 2 of Algorithm 4.3 is chosen, $UJI(i, i+l)$ is calculated as

$$1 + y(|\delta JI(i, i+q)|_{i+q}, \lceil \frac{|C_{i+q}|}{BT_f} \rceil, |C_{i+q}|)$$
$$+y(|\delta JI(i, i+q)|_{i+q}, ||JI(i+q, i+l)||, |C_{i+q}|).$$

As it is noticed that there could be more than one choice of updating $JI(i, i+l)$ in step 2 of Algorithm 4.3, p or q is chosen such that the cost of updating $JI(i, i+l)$, i.e., $UJI(i, i+l)$ is the minimum. The third part is the cost of inserting or deleting $\delta JI(i, i+l)$ into or from $JI(i, i+l)$. The cost of updating the forward $JI(i, i+l)$ is

$$1 + y(|\delta JI(i, i+l)|_i, \lceil \frac{|C_i|}{BT_f} \rceil, |C_i|)$$
$$+2 * y(|\delta JI(i, i+l)|_i, ||JI(i, i+l)||, |C_i|).$$

The cost of updating the backward $JI(i, i+l)$ is similar. The way of calculating the update cost for a complete join index hierarchy structure is similar to that of a partial join index hierarchy structure. The update cost for a base join index hierarchy structure includes either deleting or inserting $\delta JI(k, k+1)$ from or to $JI(k, k+1)$.

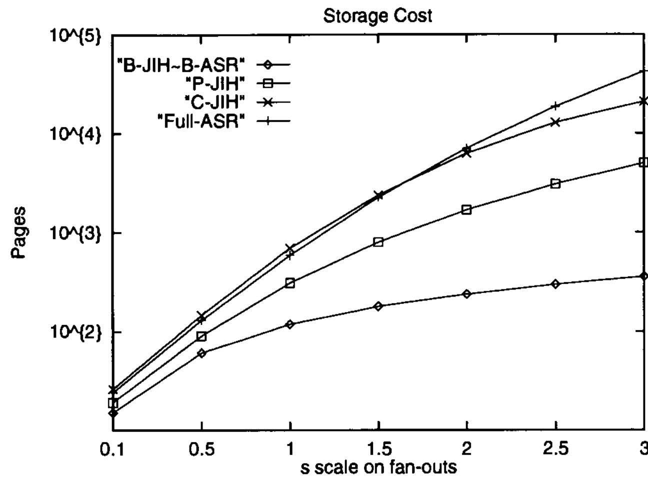


Figure 4.6: Storage Costs of B-JIH, P-JIH, C-JIH and Full-ASR vs. Fan-outs.

4.4.3 Explanation of Performance Results

The performance is conducted in the two group experiments for the index structures supporting navigations and associative searches. In the first one, four data structures, which support navigations, are compared in our performance study: (1) C-JIH as shown in Figure 4.3(a); (2) B-JIH as shown in Figure 4.3(b); (3) P-JIH as shown in Figure 4.3(c); and (4) Full-ASR (full *access support relation*), which stores the full sequences of object identifiers of the path (of length 5) in one full *access support relation.* Notice that cases (2) and (4) correspond to two extreme cases of the *access support relation* method proposed in [68], in which the former (case 2) decomposes each class pair into one component (i.e., binary decomposition of a full ASR, thus, a B-JIH is labeled B-JIH/B-ASR in the performance curves.), whereas the latter (case 4) merges the access path (sequence) into one relation.

The fan-out factors (join selectivities) is taken as the x-axis variable in Figures 4.6, 4.7, 4.8, 4.11, 4.13 and 4.14 because the performance is sensitive to the increase of the fan-out factors (join selectivities), which matches our expectation and experimentation. The set of class sizes, fan-out values, and scale changes in the analysis are in Table 4.2. The scale change factor s is introduced so that the performance under varying fan-outs can be presented in one graph. Other database parameters are set

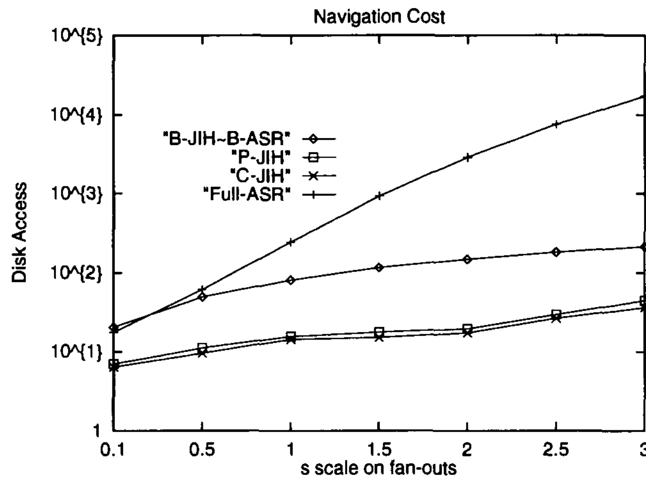


Figure 4.7: Navigation Costs of B-JIH, P-JIH, C-JIH and Full-ASR vs. Fan-outs.

to the default values as shown in Table 4.1.

Figure 4.6 shows that the storage costs increase as the fan-outs do. Full-ASR stores all the sequences of object identifiers in complete or incomplete paths. P-JIH materializes some higher level join indices of the join index hierarchies; whereas C-JIH materializes all of the higher level join indices. These are reflected in the storage cost graph. Obviously, the storage sizes of Full-ASR, P-JIH and C-JIH increase faster than that of B-JIH/B-ASR.

Figure 4.7 presents how the navigation costs increase as the fan-outs grow. It is assumed that the forward and backward counts 50% and 50% in the total cost of the navigation respectively. The navigations between C_0 and C_5, C_0 and C_4, and C_2 and C_5 weigh 20%, 40% and 40% in the total cost respectively. Notice that the navigation between C_0 and C_5 is not supported directly in the chosen P-JIH. The selectivity of navigation starting point is fixed as follows. If the navigation starts at C_i, the selectivity is chosen to be $sel * \frac{|C_0|}{|C_i|}$ where sel is the selectivity of the navigation starting at C_0. Here sel is set at 0.01, therefore, every navigation starts with 10 objects. P-JIH and C-JIH perform much better than B-JIH/B-ASR and Full-ASR. Full-ASR has the poorest performance because the whole ASR has to be retrieved (the relation is usually sorted on both head and tail classes to facilitate retrieval from

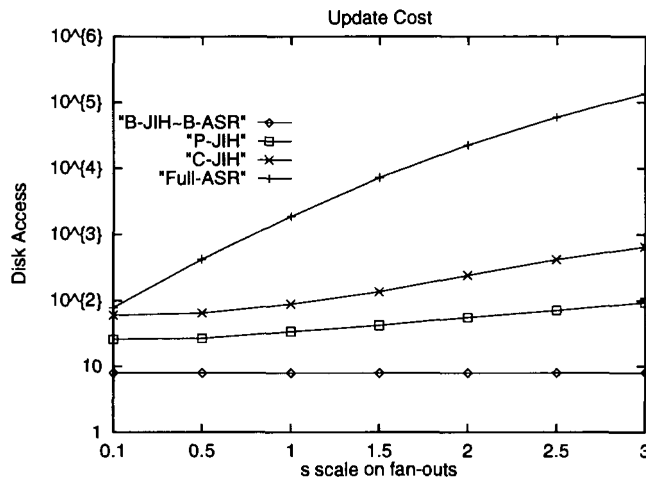


Figure 4.8: Update Costs of B-JIH, P-JIH, C-JIH and Full-ASR vs. Fan-outs.

the starting and the end points) when the navigations other than the one between head and tail classes are required.

Figure 4.8 illustrates the update costs. It is assumed that the update probability of all the base join indices are equal. Obviously, B-JIH/B-ASR has the lowest update overhead since each time only base join indices need to be updated. The update cost of Full-ASR is higher than those of other index structures and grows faster.

Figure 4.9 describes the cost of navigation and update operation mix. The total cost is defined as

$$(1 - p) * NavigationCost + p * UpdateCost,$$

where p is the update probability, and $p = 0.2$ means that there are 20% probability of updates and 80% probability of navigations among all the operations. The scale s on fan-out is set to be 1.0. With less frequent update (update probability less than 0.4), the overall performance of P-JIH and C-JIH is much better than that of B-JIH/B-ASR. All the three structures perform better than Full-ASR.

Figure 4.10 presents the navigation costs vs. navigation selectivities. The scale s on fan-outs is set to be 1.0. The selectivity at C_0 is set from 0.001 to 0.5. The navigation cost grows as the navigation selectivity increases.

Figure 4.11 presents the storage requirements vs. large fan-outs. The reason that

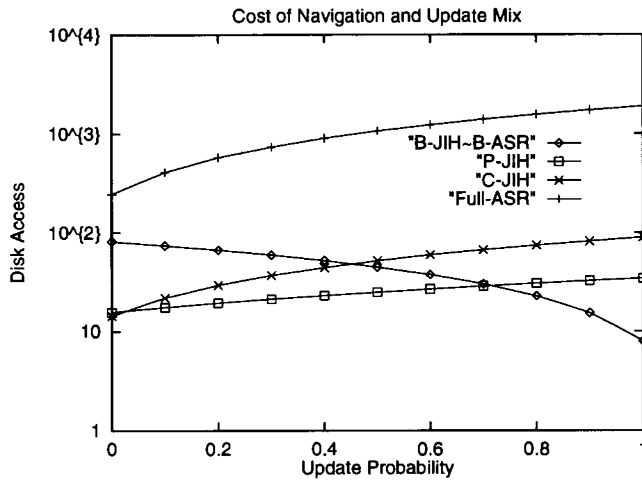


Figure 4.9: Costs of Navigation and Update mix for B-JIH, P-JIH, C-JIH and Full-ASR.

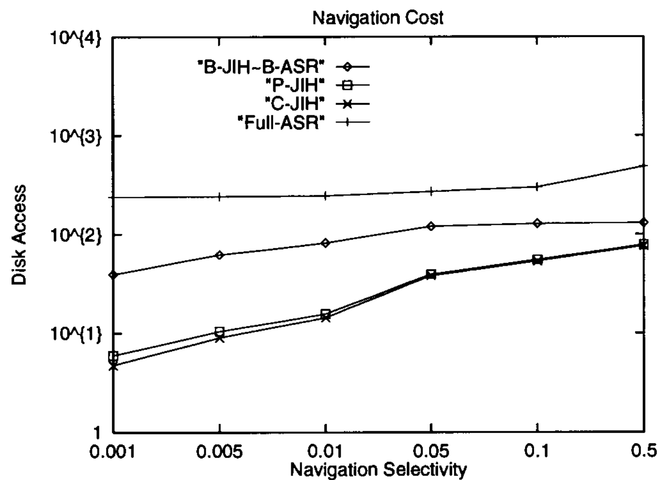


Figure 4.10: Navigation Costs of B-JIH, P-JIH, C-JIH and Full-ASR vs. Navigation Selectivities.

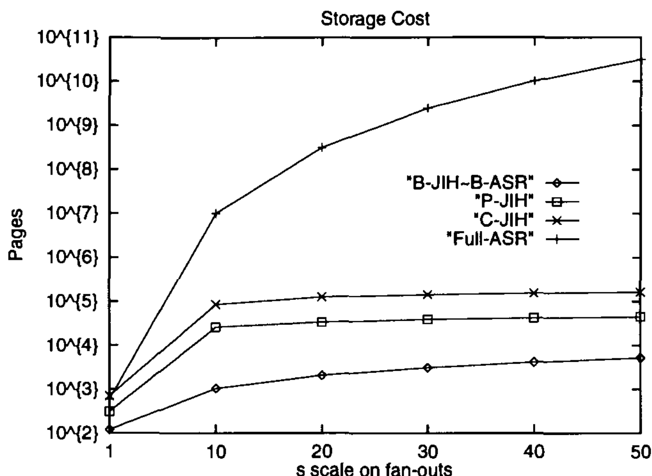


Figure 4.11: Storage Explosion with Large Fan-outs.

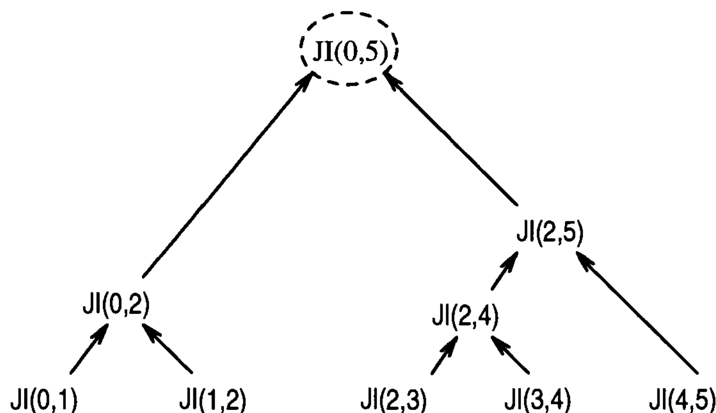


Figure 4.12: Partial Join Index for Supporting JI(0,5).

only large fan-outs are analyzed but not large cardinalities of classes is because our other performance results shows that the costs of storage, navigation and updates do not grow very fast as the cardinalities of classes increase. As one can predict, the storage cost (and hence the navigation and update costs) grows rapidly when the fan-out ratio grows. Full-ASR has the highest storage cost since multiple access paths from C_{i-1} to C_i will have to be multiplexed when pairing with the objects in C_{i+1}, etc. This also suggests that the fan-outs should be considered as an important factor for setting "fire walls" to avoid cost explosion.

In the second group experiment, five index structures, which support associative

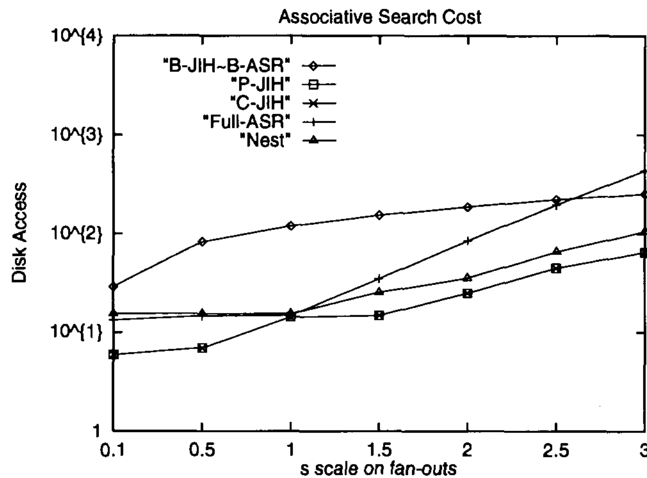


Figure 4.13: Associative Search Costs of B-JIH, P-JIH, C-JIH, Full-ASR and Nested Index vs. Fan-outs.

searches, are compared: (1) C-JIH as shown in Figure 4.3(a); (2) B-JIH as shown in Figure 4.3(b); (3) P-JIH as shown in Figure 4.12; (4) Full-ASR; and (5) Nest which denotes the nested index in [15, 14]. Notice that the target node is $JI(0,5)$ in the partial join index hierarchy in Figure 4.12.

Figure 4.13 presents how the associative search costs increase as the fan-outs grow (only the backward navigation between C_0 and C_5 is considered.). Since P-JIH and C-JIH support $JI(0,5)$ directly, their associative search costs are the same. This is indicated by the overlap of their performance curves. P-JIH and C-JIH perform better than Nest since the root node JI(0,5) of P-JIH and C-JIH is smaller than the nested index.

Figure 4.14 illustrates the update costs. When the fan-outs are small, the update costs of P-JIH and C-JIH are higher than that of the nested index. This reflects the fact that P-JIH and C-JIH maintain two copies for both forward and backward navigations while the nested index structure only keeps one (backward) copy and can only be employed for associative search (backward navigation). It is significant, however, that the update cost of the nested index grows faster than those of P-JIH and C-JIH, and exceeds them when the fan-outs become large (fan-outs scale $s > 1.5$). In a nested index, the reference information in a path has to be retrieved iteratively

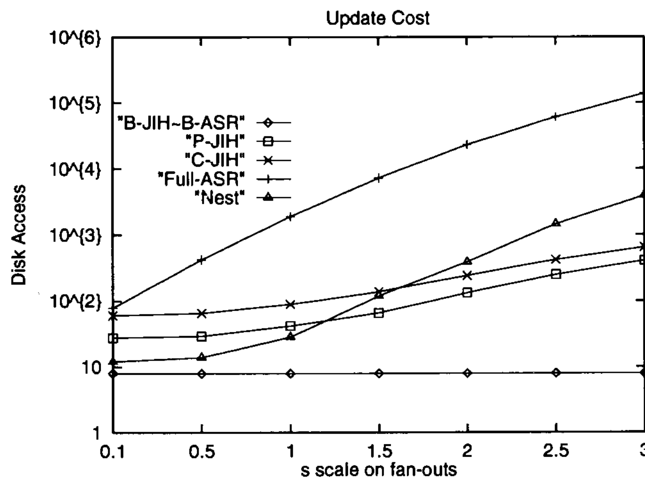


Figure 4.14: Update Costs of B-JIH, P-JIH, C-JIH, Full-ASR and Nested Index vs. Fan-outs.

from the auxiliary index so that all the appropriate records in the primary index can be updated accordingly.

In summary, the performance study shows that both P-JIH and C-JIH outperform B-JIH/B-ASR, Full-ASR and Nest in navigation, associative search and overall performance. P-JIH has better storage and better update costs than C-JIH. Clearly, join index hierarchy, especially the partial one, provides an interesting data structure to support efficient navigations in object-oriented databases.

4.5 Discussion

4.5.1 Join Index Hierarchy Which Supports Other Kinds of Navigations

The join index hierarchies discussed in the previous sections are designed for support of class composition hierarchies, i.e., navigations through a sequence of object classes via their attribute relationships. Similar join index hierarchies can be applied to support of navigations through class/subclass hierarchies, or through a sequence of classes via the relationships specified by methods and/or deduction rules.

In a schema path involving class/subclass hierarchies, if a set of subclasses associated with the same higher-level class have similar kinds of attributes, it could be beneficial to construct one (combined) base join index node instead of a large number of small join index files. This is in the same spirit of Kim, Kim and Dale [78] and Bertino [14]. For example, in Figure 4.1, the class *STUDENT* contains two subclasses: *UGRAD* and *GRAD*; whereas the latter in turn contains two subclasses: *PhDGrad* and *MasterGrad*. It is more beneficial to construct one (combined) base join index *JI(STUDENT, COURSE)* instead of three join indices: *JI(PhDGrad, COURSE)*, *JI(MasterGrad, COURSE)* and *JI(UGRAD, COURSE)*. However, if a set of subclasses, associated with the same higher level class, contain a relatively large number of objects with different kinds of class components, it could be more efficient to construct several join index files. For example, in a university database, *STUDENT*, *PROFESSOR*, and *SECRETARY* may belong to the same higher level class *PERSON*. Since each subclass could be large and different subclasses usually have quite different kinds of attributes and methods, it could be more efficient to construct different join index nodes for these subclasses.

In a schema path, same class names are allowed to appear more than once. Therefore, the corresponding join index hierarchy will support navigations with loops. This indicates that join index hierarchies can also be used to process transitive closures, a special case of linear recursions. Chapter 6 will discuss DOOD linear recursion processing.

Furthermore, there may exist more than one semantic linkage between two object classes. For example, a professor may *teach* a student (in a course), *supervise* a student (on research work), or *hire* a student (for some programming job). Thus, there may exist three kinds of semantic linkages between *PROFESSOR* and *STUDENT* in this database. A join index node is for a particular kind of semantic association which cannot be mixed up with other kinds of semantic linkages since they carry different semantics. The schema paths should be stored in the schema (data dictionary) with the identification (such as by labeling) of each semantic linkage for each join index node.

Some relationships between different classes of objects may not be specified by

existing attributes but by deduction rules or computational methods. For example, the voting eligibility of a stockholder could be defined by deduction rules based on his/her current shares of stocks, the stock holding history, etc. Thus, the linkage between the two classes, *STOCKHOLDER* and *VOTER*, are defined by rules and instantiated by rule evaluation. Similarly, the relationships between the objects in two classes, *PARK* and *LAKE*, could be specified by a spatial computational routine, which computes, based on a geographic map, whether one is inside the other, or whether two intersect, or their shortest (or highway) distances, otherwise.

The method- or deduction rule- specified object linkage can be constructed using the structure of join index hierarchy as well, by evaluation of the method/rule at the join index construction time rather than at the query processing time.

One advantage of the construction of join indices for rule- or method- defined object linkages could be the transformation of the expensive rule/method computation from query evaluation time to join index construction time. Since a method or a rule may involve recursion or iterative computation of a relatively large number of complex (such as spatial) objects, it could be quite expensive to perform such computation at the query processing time. The evaluation of such linkages at the join index construction time and the storage of the join indices together with other frequently used information (such as distance, etc. [92]) in join indices will trade storage space for query evaluation efficiency. It will be especially beneficial if such computation must be performed repeatedly or iteratively.

Furthermore, by storage of important information in join indices, some queries, especially those involving traversing in the direction in reverse to those specified in the methods or rules, can be answered efficiently. For example, to find all the lake and park pairs whose intersected regions greater than 1 square kilometer, one can retrieve the join indices and return the results directly (if the information-associated join indices [92] are constructed and the area of intersection is the associated information). However, it is impossible to compute a region from an area based on the same method which defines only the computation of an area from a geographic object but not in reverse.

4.5.2 "Fire Walls" in the Construction of Join Index Hierarchies

There may exist long object referencing sequences in queries, and any object class may serve as the starting point in a sequence of object referencing. Nevertheless, this does not suggest the construction of join index hierarchies on a very long sequence of a schema path because of the size of such a hierarchy and the cost of updates. Therefore, it is often necessary to partition a long schema path into a few short ones, or prohibitive to build some join indices or merge them into join index hierarchies.

A class linkage (by either attribute relationship, methods, or rules) which is not suitable for constructing join indices or for being merged into a join index hierarchy is called the "fire wall" of the hierarchy. It is important to identify fire walls and partition a long schema path into a set of smaller ones for the construction of easily accessible or updatable join index hierarchies.

"Fire walls" are suggested to set in the following places in the design of a join index hierarchy.

1. *Rarely referenced class linkages*: Some class linkages, though referable, are rarely used in applications, based on the examination of a relatively long history of referencing patterns. It is relatively safe to set up a fire wall at a rarely referenced point since it is fair to let rarely used referencing pay a little higher cost in accessing.

2. *Large join selectivities*: A large join selectivity implies a potentially large (or huge) join index relation. The further construction of upper level join indices would usually result in large join index relations as well. The break of the chain at this point may contribute to a relatively small join index relation and/or hierarchy.

3. *Frequently updated or multiple-source class linkages*: Some join indices may sustain frequent updates or be derived from multiple objects, classes or class relationships (such as, those computed using multiple objects or classes by methods). Such kind of class linkages may need frequent or sophisticated updates,

and update propagation to upper level join indices will likely be costly and thus it could be beneficial to set up "fire walls" there.

4.6 Summary

A join index hierarchy approach has been proposed and investigated here for efficient navigation through a sequence of object classes. The join index hierarchy organizes a set of (direct and indirect) join index nodes into a hierarchy. Three kinds of join index hierarchies are proposed and studied. Our analysis and performance study show that partial join index hierarchy has reasonably small space and update overheads, and speeds up query processing considerably in both forward and backward navigations.

Join index hierarchy is an interesting indexing structure which could be a promising candidate at solving "pointer chasing" problems in DOOD query processing. It would be interesting to compare and/or integrate the join index hierarchy method with other object optimization techniques, such as read-ahead buffering [102] and complex object assembly [66].

Chapter 5

Optimizing Queries Including Complex Selections, Joins, Aggregations and Methods

5.1 Introduction

Navigation is essentially "pointer chasing" operation which follows object identifiers from one object to another and accesses objects in one-object-at-a-time fashion. Although a navigation operation is performed over objects along a navigation path, the navigation is confined to selective objects. Constraints on the navigation can be "pushed" inside the navigation so that only relevant objects are accessed. Following the methodology of set-oriented query evaluation in relational and deductive database systems, a navigation operation is transformed into a sequence of join operations among a collection of object classes along the navigation path. Thus, the navigation can be performed in an efficient and set-oriented manner.

Navigation is the most widely used but costly operation for exploring logical relationships among complex objects in both queries and methods. Thus, the revelation of common navigation operations between a query and a method is essential for optimizing queries with encapsulated methods, reducing redundant computations, and achieving efficient query evaluation.

The following observations motivate the optimization strategies proposed in this chapter.

- *Navigation constraints.* Navigation operations are often expressed in the form of path expressions with constraint conditions such as selections and joins. "Push selection inside join" in relational database systems, and its variation "push selection inside recursion" in deductive database systems, are the effective principles to eliminate irrelevant data before the expensive computations, such as joins and recursions, are performed. "Push constraint condition inside navigation" is similar to these principles in that it effectively excludes irrelevant objects from consideration. Unlike selection and join conditions in relational and deductive database systems, however, constraints on navigations are more complicated. Different types of constraint conditions require different kinds of optimization strategies to process navigations.

- *Set-oriented evaluation.* Object-at-a-time "pointer chasing" is transformed into set-oriented evaluation of a sequence of joins. Therefore, some join methods developed for relational database systems can be applied or extended to process navigation operations, e.g., nested loop, pointer-based join algorithms [111] and join index [123]. The transformation also facilitates both forward and backward navigations among objects via a sequence of attribute relationships, class/subclass hierarchies, and relationships specified by methods and deduction rules.

- *Common navigations.* Navigation is also the most widely used operation for exploring complex objects in a method. It frequently happens that some navigations or part of navigations are shared between a query and a method. To avoid repeated computations over the shared navigations, the navigation information in a method should be revealed. Consequently, the common navigations can be exploited to accelerate query evaluation. This revelation approach offers the advantages over black box approach and the approach of restricting methods to be coded only in query languages. The former may exclude some better

query evaluation plans into consideration while the latter limits the application of user-defined methods. In addition, the revelation approach does not need to acquire the semantic information of methods, which is difficult to obtain in practice.

5.2 Motivating Examples

Navigation constraints are often expressed in the form of selection and join conditions. The following motivating examples demonstrate that different types of the selection and join conditions require different kinds of optimization strategies[1].

Example 5.1 To find out all the students who are taking some courses offered by the department of computer science, a navigation is performed from the objects in the class *STUDENT* to the objects in the class *DEPT* via the attribute *TakeCourses*, the class *COURSE*, the attribute *Dept* and the class *DEPT*. Thus the navigation can be expressed as the following path expression

$$s.TakeCourses.Dept.Name$$

where s is a variable denoting an object in the class *STUDENT*. Since the query is interested only in the students who are taking *some computer science* courses, this navigation needs not to be performed on all objects in the above classes but is confined to some selective objects. Obviously, computer science departments are the only relevant objects in the class *DEPT*. The constraint can be expressed as a selection:

$$s.TakeCourses.Dept.Name_{\exists} = \text{``}Computer\ \ Science\text{''}.$$

A set-oriented way of evaluating the path expression (performing navigation)

$$s.TakeCourses.Dept.Name$$

[1]The sample database schema is in Appendix B.

is to calculate the implicit joins[2]

$$STUDENT \bowtie COURSE \bowtie DEPT.$$

An object $s \in STUDENT$ satisfying the above selection is an instance of the following expression

$$\pi_{(STUDENT)}(\sigma_{(s.TakeCourses.Dept.Name_{\exists}=\text{"}Computer\ \ Science\text{"})}(STUDENT$$
$$\bowtie COURSE \bowtie DEPT)).$$

An object s is an instance of the above expression if and only if there is an object sequence s, c, d such that $s \in STUDENT$, $c \in s.TakeCourses$, $d = c.Dept$ and $d.Name =$ "$Computer\ \ Science$". Therefore the above algebraic expression is equivalent to

$$\pi_{(STUDENT)}(STUDENT \bowtie COURSE \bowtie \sigma_{(d.Name=\text{"}Computer\ \ Science\text{"})}(DEPT)).$$

The selection

$$s.TakeCourses.Dept.Name_{\exists} = \text{"}Computer\ \ Science\text{"}$$

has been simplified to

$$d.Name = \text{"}Computer\ \ Science\text{"}$$

and moved inside the implicit joins and onto the class $DEPT$. By performing the selection

$$d.Name = \text{"}Computer\ \ Science\text{"}$$

on the class $DEPT$ earlier than those implicit joins, the evaluation (navigation) can be more efficient because only computer science departments need to be taken into consideration. □

[2]A implicit join is a join where two objects are joinable if one is an attribute of the other. The algebra used in this chapter is similar to ENCORE/EQUAL in Shaw and Zdonik[110]. Here π and σ stand for projection and selection operations respectively.

It seems that the migration of predicates is quite similar to "push selection inside join" in relational databases. However, there are some important differences. In Example 5.1, the selection condition is simplified and moved inside the implicit joins and onto the tail class *DEPT* of the path expression. Furthermore, such kind of simplification and movement of selection conditions can not always be performed.

Example 5.2 To find out all the students who are *only* taking courses offered by the department of computer science, the navigation constraint can be expressed as follows:

$$s.TakeCourses.Dept.Name_{\forall} = \text{“}Computer\ \ Science\text{”}$$

An object $s \in STUDENT$ satisfying the above selection is an instance of the following expression

$$\pi_{(STUDENT)}(\sigma_{(s.TakeCourses.Dept.Name_{\forall}=\text{“}Computer\ \ Science\text{”})}(STUDENT \bowtie COURSE \bowtie DEPT)).$$

An object s is an instance of the above expression if and only if for all object sequences s, c, d , if $s \in STUDENT$, $c \in s.TakeCourses$ and $d = c.Dept$, then $d.Name =$ "$Computer\ \ Science$". Therefore, in this case, whether an object s is an instance of the above expression is related to all the object sequences with the same head s. The selection

$$s.TakeCourses.Dept.Name_{\forall} = \text{“}Computer\ \ Science\text{”}$$

can not be simplified and moved inside the implicit joins as in the previous example. That is, the above expression is not equivalent to

$$\pi_{(STUDENT)}(STUDENT \bowtie COURSE \bowtie \sigma_{(d.Name=\text{“}Computer\ \ Science\text{”})}(DEPT)).$$

It is, therefore, not correct to perform the selection

$$d.Name = \text{“}Computer\ \ Science\text{”}$$

before the implicit joins. The reason is that the path expression

$$s.TakeCourses.Dept.Name$$

is a set-valued one, therefore, the path expression

$$s.TakeCourses.Dept.Name$$

may correspond to one or more instances in

$$STUDENT \bowtie COURSE \bowtie DEPT.$$

The path expression

$$s.TakeCourses.Dept.Name$$

satisfies the selection if *all* those instances satisfy the condition. □

Clearly, Examples 5.1 and 5.2 show that different types of selections require different kinds of optimization strategies.

Constraints on navigations can also appear in the form of join conditions. Similarly, different optimization strategies should be applied to different types of join conditions.

Example 5.3 To find out some professor and student pairs such that some courses taught by the professors are higher level than all the courses taken by the students, a navigation may need to be performed from an object in the class *PROF* to objects in *STUDENT* via the attribute *TeachCourses*, the class *COURSE*, the attribute *TakeCourse* (in reverse), and the class *STUDENT*. The navigation constraint condition could be expressed as a path expression comparison

$$p.TeachCourses.Number_{\exists} > s.TakeCourses.Number_{\forall}$$

which could be considered as a join predicate between the two classes *PROF* and *STUDENT*. Here p stands for a variable denoting an object in the class *PROF* and s represents a variable denoting an object in the class *STUDENT*. □

In addition, aggregation functions could appear in a selection and join condition. For example,

$$MAX(u.Depts.FacultyMembers.Age) > 70$$

and

$$MAX(p.TeachCourses.Number) > MAX(s.TakeCourses.Number).$$

The questions arise whether it is possible to perform the simplification and movement on these more complicated selection and join conditions as in the previous examples? And when and how such kinds of the simplification and movement can be performed? This chapter will answer these questions and present a framework for integrating different strategies.

Interestingly, navigation information in encapsulated methods can also be exploited for efficient query processing since there may be some sharing of navigations between user-defined methods and queries.

Example 5.4 Find all the names of students who are from a metropolis or a large country with a population of over 20,000,000, and who are only taking computer science courses and taking courses over 400 level.

$$\begin{array}{ll} SELECT & s.Name \\ FROM & sSTUDENT \\ WHERE & FromLargeCountryOrMetro(s) \\ AND & s.HomeAddress.Country.Population > 20,000,000 \\ AND & s.TakeCourse.Dept.Name_{\forall} = \text{``}Computer\ \ Science\text{''} \\ AND & s.TakeCourse.Number_{\forall} > 400. \end{array}$$

If a student s comes from a metropolis or a large country, then the method

$$FromLargeCountryOrMetro(s)$$

returns true. The factorization of common sub path expressions among the path expressions in a query can be performed. Since both

$$s.TakeCourse.Dept$$

and

$$s.TakeCourse.Number$$

share the common sub path expression

$$s.TakeCourse,$$

the two selection conditions

$$s.TakeCourse.Dept_{\forall} = \text{“}Computer\ \ Science\text{”}$$

and

$$s.TakeCourse.Number_{\forall} > 400$$

have a mutual binding on $s.TakeCourse$. The evaluation result of $s.TakeCourse$ in

$$s.TakeCourse.Dept_{\forall} = \text{“}Computer\ \ Science\text{”}$$

can be used for evaluating

$$s.TakeCourse.Number_{\forall} > 400$$

or vice versa. Furthermore, there are four path expressions in the method

$$FromLargeCountryOrMetro$$

which include[3]

$$s.HomeAddress.Country.Population,$$

$$s.HomeAddress.Country.Area,$$

$$s.HomeAddress.City.Population,$$

and

$$s.HomeAddress.City.Area.$$

Their maximal common sub path expression is

$$s.HomeAddress.$$

[3]Navigations could be expressed in other forms rather than dot expressions, e.g., function cascades. It is easy to design a parser to extract navigation information from the source codes of methods.

The selection

$$s.HomeAddress.Country.Population > 20,000,000$$

of the query also has a sub path expression $s.HomeAddress$. Therefore, the common sub path expression

$$s.HomeAddress$$

needs to be evaluated only once even though it appears in the five path expressions.

□

This chapter will also present a systematic study on the revelation of navigation information in an encapsulated method and on the factorization of shared navigations among not only those in a query but also in a method during query optimization process.

5.3 Path Expression Definition

For clarity of explanation, only navigations via attribute relationships among classes are considered in the presentation. Optimization strategies for navigations through class/subclass hierarchies and relationships specified by methods and deduction rules are discussed later.

We first introduce the definition of path expressions. The definition is quite similar to that in [94]. Intuitively, a path expression represents a navigation from an object in one class to other objects in other classes via attribute relationships on a class composition hierarchy.

Definition 5.1 Path expression. $o_0.A_1.A_2...A_n$ is a path expression associated with the classes O_0, O_1, ..., O_n if o_0 is an object of class O_0 and A_i is an attribute of O_{i-1} ranged on class O_i or set of O_i for $i = 1, 2, ..., n$.

For example, $s.HomeAddress.Country.Name$ denotes the name of the country the student s comes from. $s.TakeCourses.Dept.Name$ represents the names of the departments which offer the courses taken by the student s.

The attribute A_n can be either primitive one such as the attributes $Name$ and Age of $PERSON$ or non-primitive one such as the attribute $HomeAddress$ of $PERSON$. In this chapter, for easy presentation of the results, it is assumed that A_n's are primitive attributes without loss of generality.

Definition 5.2 Object path. $(o_0, o_1, ..., o_{n-1})$ is an object path satisfying $o_0.A_1.A_2 \ldots A_n$ if for $i = 1, 2, ..., n-1$, $o_i = o_{i-1}.A_i$ when A_i is a single-valued attribute or $o_i \in o_{i-1}.A_i$ when A_i is a set-valued attribute. o_0 is called the head and o_{n-1} the tail. The attribute A_{n-1} is called the tail non-primitive attribute or the tail attribute in short.

For example, if the student s is taking computer science course CMPT200, then (s, *cmpt*200, *cs*) is an object path satisfying

$$s.TakeCourses.Dept.Name.$$

Here *cmpt*200 and *cs* are OIDs of CMPT200 and the computer science department respectively.

Definition 5.3 Single-valued and set-valued path expression. A path expression without any set-valued attributes is defined as a single-valued path expression, otherwise as a set-valued path expression. If $o_0.A_1.A_2 ... A_n$ is a set-valued path expression and A_k is a set-valued attribute and all A_i $(i < k)$ are single-valued attributes, then A_k is called the first set-valued attribute of the path expression and the classes $O_{k-1}, ..., O_{n-1}$ are called the ending non-primitive classes or the ending classes in short.

For example,

$$s.HomeAddress.Country.Name$$

is a single-valued path expression.

$$s.TakeCourses.Dept.Name$$

is a set-valued path expression and $TakeCourses$ is the first set-valued attribute.

Obviously, at most one object path may satisfy a single-valued path expression and more than one object path may satisfy a set-valued path expression. For example, a student may take more than one course, therefore, there may be more than one object path satisfying $s.TakeCourses.Dept.Name$.

Definition 5.4 Value of a path expression. If $o.A_1...A_n$ is a single-valued path expression, the value of the path expression $o.A_1...A_n$ is $o_{n-1}.A_n$ where $(o, o_1, ..., o_{n-1})$ is an object path satisfying $o.A_1...A_n$. If $o.A_1...A_n$ is a set-valued path expression, the valued of the path expression $o.A_1...A_n$ is

$$\{v|(o, o_1, ..., o_{n-1}) \text{ satisfies } o.A_1...A_n,$$

$$v = o_{n-1}.A_n \text{ if } A_n \text{ is a single-valued attribute or}$$

$$v \in o_{n-1}.A_n \text{ if } A_n \text{ is a set-valued attribute}\}.$$

For example, if the student s comes from Canada, then the value of

$$s.HomeAddress.Country.Name$$

is "Canada". If the student s is only taking the courses offered by the department of computer science and the department of mathematics, then the value of

$$s.TakeCourses.Dept.Name$$

is $\{\text{"}Computer\ \ Science\text{"}, \text{"}Mathematics\text{"}\}$.

The evaluation of $o_0.A_1...A_n$ or the navigation from o_0 via the attributes A_i, involves n object references. If these objects are not stored in a same block on a disk, then many disk accesses may be needed. Usually, given a collection of object O_0, the evaluation of $o_0.A_1...A_n$ for each $o_0 \in O_0$ are needed. A set-oriented evaluation strategy is to calculate the implicit joins

$$O_0 \bowtie O_1 \bowtie ... \bowtie O_{n-1}$$

whose instances are the object paths satisfying $o_0.A_1...A_n$ and $o_0 \in O$. Therefore, the evaluation of $o_0.A_1...A_n$ for $o_0 \in O$ is transformed into the evaluation of the implicit joins. The next two sections will identify constraints conditions associated with path expressions and show how these constraints can be employed effectively to optimize navigations.

5.4 Optimization of Complex Selections and Joins

5.4.1 Path Expression-Based Selections

Since the values of path expressions may be sets, the quantifiers $\forall$ and $\exists$ and the aggregation functions such as MIN, MAX, $COUNT$, AVG and SUM need to be introduced in operations containing path expressions [73].

Definition 5.5 Path expression selection. Path expression selections have the following form

$$f(s)_{q_1}\theta c_{q_2}$$

where s is a path expression and c is a constant or constant set, $\theta \in \{=, \neq, <, >, \leq, \geq\}$, and $f \in \{I, MAX, MIN, COUNT, AVG, SUM\}$[4]. If s is a single-valued path expression or $f \in \{MAX, MIN, COUNT, AVG, SUM\}$, q_1 can be $\exists$ or $\forall$. If c is a constant, q_2 can be $\exists$ or $\forall$. In both cases, q_1 and q_2 are always chosen to be $\exists$ by default. If $f = I$, then the path expression selection is defined as type $\theta(q_1)$ path expression selection. If $f \in \{MAX, MIN, COUNT, AVG, SUM\}$ and s is a set-valued path expression, the path expression selection is defined as type $\theta(\forall)$ path expression selection.

Obviously, by definition, selections with aggregation functions are of type $\theta(\forall)$ because the path expressions are set-valued and the aggregation functions are applied on these set-valued path expressions. Later we will show that some of them can be transformed into selections of type $\theta(\exists)$ without aggregation functions. The following path expression

$$s.TakeCourses.Dept.Name_{\forall} = \text{“}Computer\ \ Science\text{”}$$

represents that the student s is only taking computer science courses. It is a path expression selection of type $= (\forall)$.

[4] I is an identity function and used for easy presentation of the theorem with the aggregation functions MAX, MIN, COUNT, AVG and SUM

The evaluation of a path expression can be transformed into the implicit joins of its associated classes. The evaluation of a path expression selection is performed by applying the selection to the implicit joins. A natural question is when and how the selection should be "pushed"? The following Theorem 5.1 shows that different optimization strategies should be applied to different types of selections.

Theorem 5.1 $f(o.A_1...A_n)_{q_1}\theta c_{q_2}$ is a path expression selection

$$\{o|f(o.A_1...A_n)_{q_1}\theta c_{q_2},\ \ o \in O_0\} = \pi_{(O_0)}(\sigma_{(f(o.A_1...A_n)_{q_1}\theta c_{q_2})}(O_0 \bowtie \cdots \bowtie O_{n-1})).$$

If the selection is of type $\theta(\exists)$

$$\{o|f(o.A_1...A_n)_{q_1}\theta c_{q_2},\ \ o \in O_0\} = \pi_{(O_0)}(O_0 \bowtie \cdots \bowtie O_{n-2} \bowtie \sigma_{(f(o_{n-1}.A_n)_{q_1}\theta c_{q_2})}(O_{n-1})).$$

If the selection is of type $\theta(\forall)$

$$\{o|f(o.A_1...A_n)_{q_1}\theta c_{q_2},\ \ o \in O_0\} = \pi_{(O_0)}(O_0 \bowtie \cdots \bowtie O_{k-2}$$

$$\bowtie \sigma_{(f(o_{k-1}.A_k...A_n)_{q_1}\theta c_{q_2})}(O_{k-1} \bowtie \cdots \bowtie O_{n-1}))$$

where $o.A_1...A_n$ is a path expression associated with the classes $O_0, ..., O_n$, A_k is the first set-valued attribute, $o_{k-1} \in O_{k-1}$, $f \in \{I, MAX, MIN, COUNT, AVG, SUM\}$ and $\theta \in \{=, \neq, <, >, \leq, \geq\}$.
Proof: see Appendix C. □

Example 5.5 Let us consider Example 5.1 and 5.2 again.

$$s.TakeCourses.Dept.Name_{\exists} = \text{"}Computer\ \ Science\text{"}$$

represents that the student s is taking some computer science courses. It is a path expression selection of type $= (\exists)$.

$$\{s|s.TakeCourses.Dept.Name_{\exists} = \text{"}Computer\ \ Science\text{"},\ \ s \in STUDENT\}$$

$$= \pi_{(STUDENT)}(STUDENT \bowtie COURSE \bowtie \sigma_{(d.Name=\text{"}Computer\ \ Science\text{"})}(DEPT)).$$

And

$$s.TakeCourses.Dept.Name_{\forall} = \text{"}ComputerScience\text{"}$$

represents that the student s is only taking computer science courses. It is a path expression selection of type $= (\forall)$.

$$\{s | s.TakeCourses.Dept.Name_{\forall} = \text{“}Computer\ \ Science\text{”},\ \ s \in STUDENT\}$$

$$= \pi_{(STUDENT)}(\sigma_{(s.TakeCourses.Dept.Name_{\forall}=\text{“}Computer\ \ Science\text{”})}(STUDENT \bowtie COURSE \bowtie DEPT)).$$

□

The above examples show when and how the path expression selections can be simplified and moved inside the implicit joins of the classes associated with the path expressions.

The following example illustrates when and how a selection with an aggregation function can be simplified and moved inside the implicit joins of the classes associated with the path expression.

Example 5.6 $COUNT(p.Dept.OfferPrograms) > 3$ denotes that the department the professor p is working for offers more than three programs.

$$\{p | COUNT(p.Dept.OfferPrograms) > 3,\ \ p \in PROF\}$$

$$= \pi_{(PROF)}(PROF \bowtie \sigma_{(COUNT(d.OfferPrograms)>3)}(DEPT \bowtie PROGRAM)).$$

□

Here the selection condition migration is similar to “push selection inside join” which usually generates efficient query evaluation in relational query processing. But it is different from “push selection inside join” in relational query processing. If a path expression selection is of type $\theta(\exists)$, the selection condition can be simplified and moved onto the range class of the tail attribute of the path expression. If a path expression selection is of type $\theta(\forall)$, then the selection condition can only be simplified and moved onto the implicit joins of the ending classes of the path expression. We have the following observation:

Observation 5.1 Theorem 5.1 can be summarized as follows:

1. If a path expression selection is of type $\theta(\exists)$, then the selection condition can be simplified and moved onto the range class of the tail attribute of the path expression.

2. If a path expression selection is of type $\theta(\forall)$, then the selection condition can be simplified and moved onto the implicit joins of the ending classes of the path expression.

Theorem 5.2 If s is a set-valued path expression, then

$$s_\exists > c \Longleftrightarrow MAX(s) > c,$$

$$s_\forall > c \Longleftrightarrow MIN(s) > c.$$

Similar results hold for other comparisons such as $\geq$, $<$, and $\leq$.
Proof: The proof is easy and omitted. □

This theorem illuminates the relationship between selections with and without aggregation functions. $MAX(s) > c$, which is a selection of type $> (\forall)$, can be translated into $s_\exists > c$, which is a selection of type $> (\exists)$. Obviously, the later one can be processed more efficiently than the former. For example,

$$MAX(u.Depts.FacultyMembers.Age) > 70$$

denotes that the oldest professor in the university u is over 70. It is equivalent to

$$u.Depts.FacultyMembers.Age_\exists > 70$$

which is a selection of type $> (\exists)$.

5.4.2 Path Expression-Based Joins

Constraints on navigations can also be present in the form of join conditions which have more general form than those in the relational databases and include comparisons operations between two path expressions.

Definition 5.6 Path expression comparison. Path expression comparisons have the following form

$$f(s)_{q_1}\theta g(t)_{q_2}$$

where $q_i \in \{\forall, \exists\}$, s and t are path expressions, $\theta \in \{=, \neq, <, >, \leq, \geq, \subseteq, \subset, \supseteq, \supset\}$ and $f, g \in \{I, MAX, MIN, COUNT, AVG, SUM\}$. If s is a single-valued path expression or $f \in \{MAX, MIN, COUNT, AVG, SUM\}$, q_1 can be either $\exists$ or $\forall$. If t is a single-valued path expression or $g \in \{MAX, MIN, COUNT, AVG, SUM\}$, q_2 can be either $\exists$ or $\forall$. In both cases, q_1 and q_2 are always chosen to be $\exists$ by default. If both $f = I$ and $g = I$, the comparison is defined as type $\theta(q_1, q_2)$. If $f = I$ and $g \in \{MAX, MIN, COUNT, AVG, SUM\}$, the comparison is defined as type $\theta(q_1, \forall)$. If $f \in \{MAX, MIN, COUNT, AVG, SUM\}$ and $g = I$, the comparison is defined as type $\theta(\forall, q_2)$. If $f, g \in \{MAX, MIN, COUNT, AVG, SUM\}$ and s and t are set-valued path expressions, the comparison is defined as type $\theta(\forall, \forall)$. Only when s and t are set-valued path expressions, and f and g are identity functions then θ could be one of $\{\subseteq, \subset, \supseteq, \supset\}$. In these cases, the comparison is defined as type $\theta(\forall, \forall)$.

For example,

$$u.President.Age > d.FacultyMembers.Age_{\forall}$$

denotes that the president of the university u is older than any faculty member of the department d. The comparison is of type $> (\exists, \forall)$.

The evaluation of the path expression comparison between

$$o.A_1...A_n$$

and

$$o'.B_1...B_m$$

is transformed into a join between

$$O_0 \bowtie \cdots \bowtie O_{n-1} \ \ and \ \ O'_0 \bowtie \cdots \bowtie O'_{m-1}$$

and the comparison operation becomes the join conditions.

Definition 5.7 Path expression join. A join corresponding to a path expression comparison is called a path expression join.

A natural question arises as in the evaluation of selections. When and how join conditions can be simplified and moved inside the implicit joins of the classes associated with the path expressions? The following Theorem 5.3 answers the question.

Theorem 5.3 $f(o.A_1...A_n)_{q_1}\theta g(o'.B_1...B_m)_{q_2}$ is a path expression comparison, we have

$$\{(o,o')|f(o.A_1...A_n)_{q_1}\theta g(o'.B_1...B_m)_{q_2},\ \ o\in O_0,\ \ o'\in O_0'\}$$
$$=\pi_{(O_0,O_0')}(\sigma_{(f(o.A_1...A_n)_{q_1}\theta g(o'.B_1...B_m)_{q_2})}((O_0\bowtie...\bowtie O_{n-1})\times(O_{m-1}'\bowtie...\bowtie O_0'))).$$

1. If the path expression comparison is of type $\theta(\exists,\exists)$, then

$$\{(o,o')|f(o.A_1...A_n)_{q_1}\theta g(o'.B_1...B_m)_{q_2},\ \ o\in O_0,\ \ o'\in O_0'\}$$
$$=\pi_{(O_0,O_0')}(O_0\bowtie\cdots\bowtie\sigma_{(f(o_{n-1}.A_n)_{q_1}\theta g(o_{m-1}'.B_m)_{q_2})}(O_{n-1}\times O_{m-1}')\bowtie\cdots\bowtie O_0').$$

2. If the path expression comparison is of type $\theta(\exists,\forall)$, then

$$\{(o,o')|f(o.A_1...A_n)_{q_1}\theta g(o'.B_1...B_m)_{q_2},\ \ o\in O_0,\ \ o'\in O_0'\}$$
$$=\pi_{(O_0,O_0')}(O_0\bowtie\cdots\bowtie\sigma_{(f(o_{n-1}.A_n)_{q_1}\theta g(o_{k-1}'.B_k...B_m)_{q_2})}(O_{n-1}$$
$$\times(O_{m-1}'\bowtie\cdots\bowtie O_{k-1}'))\bowtie\cdots\bowtie O_0').$$

3. If the path expression comparison is of type $\theta(\forall,\exists)$ or $\theta(\forall,\forall)$, then

$$\{(o,o')|f(o.A_1...A_n)_{q_1}\theta g(o'.B_1...B_m)_{q_2},\ \ o\in O_0,\ \ o'\in O_0'\}$$
$$=\pi_{(O_0,O_0')}(O_0\bowtie\cdots\bowtie O_{j-2}\bowtie\sigma_{(f(o_{j-1}.A_j...A_n)_{q_1}\theta g(o_{k-1}'.B_k...B_m)_{q_2})}((O_{j-1}\bowtie\cdots\bowtie O_{n-1})$$
$$\times(O_{m-1}'\bowtie\cdots\bowtie O_{k-1}'))\bowtie O_{k-2}'\bowtie\cdots\bowtie O_0')$$

where $o.A_1...A_n$ and $o'.B_1...B_m$ are path expressions associated with the classes $O_0,...,O_n$ and $O_0',...,O_m'$ respectively, A_j is the first set-valued attribute of $o.A_1...A_n$ and B_k is the first set-valued attribute of $o'.B_1...B_m$, $o_i\in O_i$, $o_i'\in O_i'$ $q_i\in\{\forall,\exists\}$, and $\theta\in\{=,\neq,<,>,\leq,\geq,\subseteq,\subset,\supseteq,\supset\}$. $f,g\in\{I,MAX,MIN,COUNT,AVG,SUM\}$.
Proof: The proof is similar to that of Theorem 5.1. □

Example 5.7

$$p.TeachCourses.Number_{\exists} > s.TakeCourses.Number_{\forall}$$

is a path expression join of type $> (\exists, \forall)$ which denotes some courses taught by the professor p are higher level than all the courses taken by the student s.

$$\{(p, s) | p.TeachCourses.Number_{\exists} > s.TakeCourses.Number_{\forall},$$
$$p \in PROF, \quad s \in STUDENT\}$$
$$= \pi_{(PROF,STUDENT)}(PROF \bowtie$$
$$\sigma_{(c.Number > s.TakeCourses.Number_{\forall})}(COURSE \times (COURSE' \bowtie STUDENT))).$$

□

We have the following observation:

Observation 5.2 Theorem 5.3 can be summarized as follows:

1. If a path expression join is of type $\theta(\exists, \exists)$, the join condition can be simplified and moved onto the join of the range classes of the tail attributes of the two path expressions.

2. If a set-valued join is of type $\theta(\exists, \forall)$, then the join condition can be simplified and moved onto the join of the class of the tail attribute of the left path expression and the ending classes of the right path expression.

3. If a set-valued join is of type $\theta(\forall, \exists)$ or $\theta(\forall, \forall)$, then the join condition can be simplified and moved onto the join of the ending classes of both the path expressions.

Theorem 5.4 If s and t are two set-valued path expressions, then

$$s_{\exists} > t_{\exists} \Longleftrightarrow MAX(s) > t_{\exists} \Longleftrightarrow s_{\exists} > MIN(t) \Longleftrightarrow MAX(s) > MIN(t)$$
$$s_{\exists} > t_{\forall} \Longleftrightarrow MAX(s) > t_{\forall} \Longleftrightarrow s_{\exists} > MAX(t) \Longleftrightarrow MAX(s) > MAX(t)$$

$$s_\forall > t_\exists \Longleftrightarrow MIN(s) > t_\exists \Longleftrightarrow s_\forall > MIN(t) \Longleftrightarrow MIN(s) > MIN(t)$$

$$s_\forall > t_\forall \Longleftrightarrow MIN(s) > t_\forall \Longleftrightarrow s_\forall > MAX(t) \Longleftrightarrow MIN(s) > MAX(t)$$

Similar results hold for other comparisons such as $\geq$, $<$, and $\leq$.

Proof: The proof is easy. □

This theorem illustrates the relationship between path expression comparisons with and without aggregation functions. $s_\exists > t_\exists$ can be processed more efficiently than the other three equivalent comparisons with aggregation functions. $s_\exists > t_\forall$ and $s_\exists > MAX(t)$ can be processed more efficiently than the other two equivalent comparisons with aggregation functions.

5.5 Classification and Cost Estimation of Methods

5.5.1 Method Definition

A method is defined as a function associated with a group of classes:

$$m : O_1 \times \cdots \times O_n \rightarrow O_{n+1}$$

The above form of the method m is equivalent to

$$m' : O_1 \times \cdots \times O_n \times O_{n+1} \rightarrow Boolean$$

Therefore, methods can appear in the same way as predicates. The usage of terms of predicates and methods will be exchanged in the rest of the chapter. Here O_1, O_2, ..., and O_{n+1} are the range classes of the arguments of the method m. For example,

$$birthday : EMPLOYEE \rightarrow DATE$$

or

$$birthday : EMPLOYEE \times DATE \rightarrow Boolean$$

where $EMPLOYEE$ is a persistent class while $DATE$ may be a non-persistent class[5].

[5]Since only objects of persistent classes are concerned, objects will refer to persistent objects without confusion in the rest of the chapter. We also make an assumption that all methods do not involve any update operations throughout the chapter.

In a method, some navigations start from argument objects while others begin with objects other than argument objects. Therefore, there are two types of path expressions in a method. *Argument path expressions* are those path expressions originating from argument objects of a method. All path expressions not originating from argument objects are called hidden path expressions. *Hidden path expressions* and path expressions in queries do not share any direct mutual bindings. Only argument path expression and path expressions in queries may possibly share mutual bindings.

Definition 5.8 Maximum common sub argument path expression. Argument path expressions can be clustered according to the argument objects where they originate. An n-persistent arguments method has n clusters of argument path expressions. For each cluster, the maximal common sub argument path expression is defined as the common sub argument path expressions of maximal length.

The method *FromLargeCountryOrMetro*, for example, has one cluster of argument path expressions:

$$\{s.HomeAddress.Country.Population,$$

$$s.HomeAddress.Country.Area,$$

$$s.HomeAddress.City.Population,$$

$$s.HomeAddress.City.Area\}$$

s.HomeAddress is its maximal common sub argument path expression.

Path expressions in a conjunctive query may have bindings on all argument path expressions in methods. However, argument path expressions in a method may not have direct bindings on path expressions in a query even though argument path expressions in a method and path expressions in a query share common sub path expressions. It is only certain that maximal common sub argument path expressions in a method may have direct bindings on path expressions in a query. For example, consider the query in Example 5.4. In the method *FromLargeCountryOrMetro*, the maximal common sub argument path expression *s.HomeAddress* has a binding on the path expression

$$s.HomeAddress.Country.Population$$

in the query. However, the argument path expression

$$s.HomeAddress.Country.Population$$

in the method does not have a binding on the path expression

$$s.HomeAddress.Country.Population$$

in the query because of *or* in the method. Let us consider the following query,

Example 5.8 Find all the names of students who are from a metropolis *and* a large country with a population of over 20,000,000, and who are only taking computer science courses and taking courses over 400 level.

$$\begin{array}{ll} SELECT & s.Name \\ FROM & sSTUDENT \\ WHERE & FromLargeCountryAndMetro(s) \\ AND & s.HomeAddress.Country.Population > 20,000,000 \\ AND & s.TakeCourse.Dept.Name_{\forall} = \text{“}Computer\ \ Science\text{''} \\ AND & s.TakeCourse.Number_{\forall} > 400 \end{array}$$

□

If a student s comes from a metropolis *and* a large country, then the method

$$FromLargeCountryAndMetro(s)$$

returns true. Then the argument path expression

$$s.HomeAddress.Country.Population$$

does have a binding on the path expression

$$s.HomeAddress.Country.Population$$

in the query because of *"and"* in the method. It is very hard to automatically acquire semantics such as relationships among path expressions in a method. However, maximal common sub argument path expressions can always be used as bindings on path expressions in queries. In this sense, methods in our approach are still not white boxes but grey ones.

5.5.2 Method Classification

A method m with n-persistent arguments can be classified according to how many of argument path expressions are in each cluster of argument path expressions and whether there are hidden path expressions in the method.

Definition 5.9 Type $n-(k_1,k_2,...,k_n)$ method. A method m is defined as type $n-(k_1,k_2,...,k_n)$ method if it has n persistent arguments and k_i denotes that there are k_i argument path expressions originating from the ith persistent argument. If m has hidden path expressions, it is defined as type $n-(k_1,k_2,...,k_n)(Y)$ method, otherwise type $n-(k_1,k_2,...,k_n)(N)$ method.

In methods of types 1-(1)(N), 1-(1)(Y), n-(1, 1, ..., 1)(N) and n-(1, 1, ..., 1)(Y), there is only one argument path expression in each cluster of argument path expressions. These argument path expressions are, therefore, maximal common sub argument path expressions.

Definition 5.10 Selection-type and join-type methods. A method of type 1-(n) is called a selection-type method. A method of type $n-(k_1,...,k_n)$, where $n>1$, is called a join-type method.

Theorem 5.5 m is a selection-type method with $o.A_1...A_n$ as its maximal common sub argument path expression.

1. If $o.A_1...A_n$ is a single-valued path expression,

$$\{o|m(o),\ \ o\in O_0\}$$

$$=\{o|m'(o.A_1...A_n),\ \ o\in O_0\}$$

$$=\pi_{(O_0)}(O_0\bowtie\cdots\bowtie\sigma_{m'(o_n)}(O_n))$$

where $m'(o.A_1...A_n)=m(o)$ and $o_n\in O_n$.

2. If $o.A_1...A_n$ is a set-valued path expression,

$$\{o|m(o),\ \ o \in O_0\}$$

$$= \{o|m'(o.A_1...A_{s-1}),\ \ o \in O_0\}$$

$$= \pi_{(O)}(O_0 \bowtie \cdots \bowtie \sigma_{m'(o_{s-1})}(O_{s-1}))$$

where A_s is the first set-valued attribute, $m'(o.A_1...A_{s-1}) = m(o)$ and $o_{s-1} \in O_{s-1}$.

Proof. The proof is in Appendix C. □

Theorem 5.6 m is a method of type $2-(k,l)$ with $o.A_1...A_p$ and $o'.B_1...B_q$ as its maximal common sub argument path expressions.

1. If both $o.A_1...A_p$ and $o'.B_1...B_q$ are single-valued path expressions,

$$\{(o,o')|m(o,o'),\ \ o \in O_0,\ \ o' \in O'_0\}$$

$$= \{(o,o')|m'(o.A_1...A_p, o'.B_1...B_q),\ \ o \in O_0,\ \ o' \in O'_0\}$$

$$= \pi_{(O_0,O'_0)}(O_0 \bowtie \cdots \bowtie \sigma_{m'(o_p,o'_q)}(O_p \bowtie O'_q) \bowtie \cdots \bowtie O'_0)$$

where $m'(o.A_1...A_p, o'.B_1...B_q) = m(o,o')$, $o_p \in O_p$ and $o'_q \in O'_q$.

2. If $o.A_1...A_p$ is a single-valued path expression and $o'.B_1...B_q$ is a set-valued path expression,

$$\{(o,o')|m(o,o'),\ \ o \in O_0,\ \ o' \in O'_0\}$$

$$= \{(o,o')|m'(o.A_1...A_p, o'.B_1...B_{t-1})\ \ o \in O_0,\ \ o' \in O'_0\}$$

$$= \pi_{(O_0,O'_0)}(O_0 \bowtie \cdots \bowtie \sigma_{m'(o_p,o'_{t-1})}(O_p \bowtie O'_{t-1}) \bowtie \cdots \bowtie O'_0)$$

where B_t is the first set-valued attribute, $m'(o.A_1...A_p, o'.B_1...B_{t-1}) = m(o,o')$, $o_p \in O_p$ and $O'_{t-1} \in O'_{t-1}$.

3. If $o.A_1...Ap$ is a set-valued path expression and $o'.B_1...B_q$ is a single-valued path expression,

$$\{(o,o')|m(o,o'),\ \ o \in O_0,\ \ o' \in O_0'\}$$
$$= \{(o,o')|m'(o.A_1...A_{s-1},o'.B_1...B_q),\ \ o \in O_0,\ \ o' \in O_0'\}$$
$$= \pi_{(O_0,O_0')}(O_0 \bowtie \cdots \bowtie \sigma_{m'(o_{s-1},o_q')}(O_{s-1} \bowtie O_q') \bowtie \cdots \bowtie O_0')$$

where A_s is the first set-valued attribute, $m'(o.A_1...A_{s-1},o'.B_1...B_q) = m(o,o')$, $o_{s-1} \in O_{s-1}$ and $o_q' \in O_q'$.

4. If both $o.A_1...A_p$ and $o'.B_1...B_q$ are set-valued path expressions,

$$\{(o,o')|m(o,o'),\ \ o \in O_0,\ \ o' \in O_0'\}$$
$$= \{(o,o')|m'(o.A_1...A_{s-1},o'.B_1...B_{t-1}),\ \ o \in O_0,\ \ o' \in O_0'\}$$
$$= \pi_{(O_0,O_0')}(O_0 \bowtie \cdots \bowtie \sigma_{m'(o_{s-1},o_{t-1}')}(O_{s-1} \bowtie O_{t-1}') \bowtie \cdots \bowtie O_0')$$

where A_s and B_t are the first set-valued attributes, $m'(o.A_1...A_{s-1},o'.B_1...B_{t-1}) = m(o,o')$, $o_{s-1} \in O_{s-1}$ and $o_{t-1}' \in O_{t-1}'$.

Proof. The proof is similar to Theorem 5.5. □

The theorems for methods of type $n-(k_1,...,k_n)$ are similar. In a method, if some maximal common sub argument path expressions are single-valued ones, then the method can be evaluated before the evaluation of the implicit joins of the classes associated with these single-valued maximal common sub argument path expressions. The evaluation of the method has to be delayed after those of the implicit joins of the ending classes associated with those set-valued maximal common sub argument path expressions.

Example 5.9

$$FromLargeCountryOrMetro(s)$$

can be rewritten as

$$FLCOM(s.HomeAddress)$$

where

```
FLCOM:ADDR → BOOLEAN
FLCOM(h:ADDR):BOOLEAN
  begin
    if (h.Country.Population > 15,000,000 and h.Country.Area > 2,000,000)
      or (h.City.Population > 1,000,000 and h.City.Area > 500)
    then return(True) else return(False)
  end
```

The method *FLCOM* can be considered as a method on *ADDR* and evaluated before the evaluation of the implicit join of the classes *STUDENT* and *ADDR* associated with the maximal common sub argument path expression $s.HomeAddress$ because

$$\{s|FromLargeCountryOrMetro(s),\ \ s \in STUDENT\}$$

$$= \pi_{STUDENT}(STUDENT \bowtie \sigma_{FLCOM(a)}(ADDR))$$

□

5.5.3 Cost Estimation of Method Evaluation

The cost of evaluating a method involves two parts: the cost involving argument path expressions *ARG* and the cost involving hidden path expressions HID. Since maximal common sub argument path expressions are revealed, they will not be included in the cost estimation of the methods. We define a unit cost as one reference of an object in a class. For example, the cost of the reference of $o.A$, where A is an attribute of o, is $C(o.A) = 1$.

- m is of type $1-(1)(N)$ or type $n-(1,1,...,1)(N)$. In this two cases, there are no hidden path expressions and the argument path expressions are themselves the maximal common sub argument path expressions. Consequently, the cost of calculating these types of methods is 0.

- m is of type $1-(1)(Y)$ or type $n-(1,1,...,1)(Y)$. Since all the argument path expressions are themselves the maximal common sub argument path expressions,

the cost of calculating these types of methods only involves the hidden path expressions.

- m is of type $1-(n)(N)$ or type $n-(k_1,...,k_n)(N)$. There are no hidden path expressions in these two types of methods. The cost of calculating these types of methods only involves argument path expressions (excluding the maximal common sub argument path expressions).
- m is of type $1-(n)(Y)$ or type $n-(k_1,...,k_n)(Y)$. In these two cases, since there are both hidden path expressions and argument path expressions, the cost of evaluating these types of methods involve both types of the path expressions (excluding the maximal common sub argument path expressions).

In the following, we will show how to estimate the cost of the method

$$Top10WellPaidUniv(u).$$

If a president of an university u is among the top 10 well paid presidents of all universities, then the method

$$Top10WellPaidUniv(u).$$

returns true. Obviously, it is of type $1-(1)(Y)$. Our approach is quite similar to Kemper et al [71]. However, the maximal common sub argument path expression, $u.President$, is excluded when the cost of evaluating the method is estimated.

$$C(Top10WellPaidUniv) = ARG + HID$$

where

$$ARG = 0$$

and

$$HID = N * C(v.President.Salary)$$

where N is the number of universities in class *UNIVERSITY*. Because the maximal common sub argument path expressions are revealed and are not included in the cost estimation, $ARG = 0$. $C(v.President.Salary) = 2$. Thus,

$$C(Top10WellPaidUniv) = 2 * N$$

In the above estimation, we do not consider any storage information such as object clustering, buffer size and object indices to avoid any complication. However, the estimation illustrates the numbers of object references, therefore, highlights the cost of evaluating a method. More advanced estimation methods are discussed later.

5.6 Query Graph and Query Plan Generation

In the previous two sections, we have considered optimization of selections and joins which include path expressions and methods. The following definitions summarize the classifications of selections and joins.

Definition 5.11 Selection. Single-valued selections include path expression selections of type $\theta(\exists)$ and selections with selection-type methods as selection conditions where their maximal common sub argument path expressions are single-valued. Set-valued selections include path expression selections of type $\theta(\forall)$ and selections with selection-type methods as selection conditions where their maximal common sub argument path expressions are set-valued. Both single-valued and set-valued selections are called selections.

Definition 5.12 Join. Single-valued joins include implicit joins, path expression joins of type $\theta(\exists, \exists)$, and joins with join-type methods as join conditions where all maximal common sub argument path expressions are single-valued. Set-valued joins include path expression joins of type $\theta(\exists, \forall)$, $\theta(\forall, \exists)$, $\theta(\forall, \forall)$ and joins with join-type methods as join conditions where some maximal common sub argument path expressions are set-valued. Both single-valued and set-valued joins are called joins.

By identifying different types of constraint conditions and applying appropriate optimization strategies, we can achieve "push constraint inside navigation". Thus inexpensive but highly selective constraints can be evaluated to eliminate irrelevant objects before costly navigation operations are performed. Common navigation operations are exploited among queries and user-defined methods by revealing the encapsulated methods.

In a path expression selection of type $\theta(\exists)$, the selection predicate can be simplified and moved to the class of tail attribute of the path expression. The selection can be performed before the implicit joins of the classes associated with the path expressions and should be done as early as possible. However, in a path expression selection of type $\theta(\forall)$, the selection has to be delayed after the evaluation of implicit joins of the ending classes associated with the path expression. In a path expression join of type $\theta(\exists, \exists)$, the explicit join can be evaluated before the implicit joins of the classes associated with both the path expressions. In a path expression join of type $\theta(\exists, \forall)$, the explicit join can only be evaluated after the implicit joins of the ending classes associated with the right path expression. In a path expression join of type $\theta(\forall, \exists)$ or $\theta(\forall, \forall)$, the explicit join can only be evaluated after the implicit joins of the ending classes associated with both the path expressions. Some selections and joins with aggregation functions can be translated into the equivalent and more efficient forms of selections and joins. In a selection-type method with a single-valued maximal common sub argument path expression, the method can be evaluated before the implicit joins of the classes associated with the maximal common sub argument path expression. In a selection-type method with a set-valued maximal common sub argument path expression, the method can only be evaluated after the implicit joins of the ending classes associated with the maximal common sub argument path expression. A join-type method can be evaluated before the implicit joins of the classes associated with single-valued maximal common sub argument path expressions and after the implicit joins of the ending classes associated with set-valued maximal common sub argument path expressions. The optimization strategies are summarized as follows:

- perform single-valued selections as early as possible;
- perform set-valued selections after implicit joins of ending classes of relevant path expressions;
- perform set-valued joins after implicit joins of ending classes of relevant path expressions; and

- perform transformation of selections and joins with aggregation functions into the equivalent and more efficient forms of selections and joins without aggregation functions.

5.6.1 Query Graph

There are some proposals of query graphs for object queries, for example, Banerjee et al. [8], Cluet and Delobel [29] and Lanzellote et al. [90]. However, their proposals do not meet all of our requirements to facilitate the implementation of the proposed optimization strategies. A query graph should be able to represent single-valued and set-valued selections, single-valued and set-valued joins as well as the factorizations of common sub path expressions among path expressions in a query.

Definition 5.13 Query graph. Query graph is a hypergraph $H = (V, E)$, where V is constructed as follows:

1. Nodes correspond to all object variables appearing in a query. Different object variables correspond to different nodes even though they are in a same class.

2. If $t.A_1.A_2...A_m$ is a path expression appearing in the query, add nodes corresponding to the range classes of $A_1, A_2, ..., A_{m-1}$ to V.

E is constructed as follows:

1. If $t.A_1.A_2...A_m$ is a path expression appearing in the query, add an edge connecting the node corresponding to object variable t and the node corresponding to the range class of the attribute A_1 and the edges connecting the two nodes representing the range classes of the attributes A_{i-1} and A_i where $2 \leq i \leq m-1$.

2. If a path expression selection appears in the query, add an one node edge which contains one node corresponding to the range class of the tail attribute of the path expression if the selection is of type $\theta(\exists)$. The edge denotes the predicate on the range class of the tail attribute. If the selection is of type $\theta(\forall)$, add a superedge connecting the nodes corresponding to the ending classes associated with the path expression.

3. If a path expression comparison appears in the query, add an edge connecting the two nodes corresponding to the range classes of the tail attributes of the two path expressions if the path expression comparison is of type $\theta(\exists, \exists)$. The edge denotes the explicit joins. If the path expression comparison is of type $\theta(\exists, \forall)$, add a superedge connecting the nodes corresponding to the range class of tail attribute of the left path expression and the ending classes associated with the right path expression. If the path expression comparison is of type $\theta(\forall, \exists)$ or $\theta(\forall, \forall)$, add a superedge connecting the nodes corresponding to the ending classes associated with both the path expressions.

4. If a method appears in the query and $t.A_1.A_2...A_n$ is a maximal common sub argument path expression, then add a superedge connecting following nodes. If $t.A_1.A_2...A_n$ is a single-valued path expression, the edge contains the node corresponding to the range class of the tail attribute only. If $t.A_1.A_2...A_n$ is a set-valued path expression, the superedge contains the nodes corresponding to the ending classes associated with the path expression.

The following examples illustrate how a query graph is constructed and query evaluation plans are generated.

Example 5.10 Finds all pairs of students and universities such that all the faculty members of the university are over 40, the total numbers of the courses offered by the universities are over 500, some courses taught by the university presidents are higher level than all those of the courses taken by the students, and the students are from the city Prince George.

$$
\begin{array}{ll}
SELECT & u.Name,\ s.Name \\
FROM & u\ UNIVERSITY,\ s\ STUDENT \\
WHERE & u.Depts.FacultyMembers.Age_{\forall} > 40 \\
AND & COUNT(u.Depts.OfferCourses) > 500 \\
AND & u.President.TeachCourses.Number_{\exists} > s.TakeCourses.Number_{\forall} \\
AND & s.HomeAddress.City.Name = \text{``Prince George''}
\end{array}
$$

□

Its query graph is in Figure 5.1. A node corresponding to an object variable or a class is represented by an ellipse. A conventional directed edge denotes an implicit join between two classes. A superedge is represented by an area which is bounded by a closed line and contains all the nodes of the superedge. For example, the one-node superedge $\{CITY\}$ denotes the single-valued selection

$$c.Name = \text{``}Prince\ \ George\text{''}$$

which is the result of the simplification and movement of the single-valued selection

$$s.HomeAddress.City.Name = \text{``}Prince\ \ George\text{''}.$$

$\{UNIVERSITY, DEPT, PROF\}$ is a superedge which denotes the set-valued selection

$$u.Depts.FacultyMembers.Age_{\forall} > 40.$$

The superedge $\{UNIVERSITY, DEPT, COURSE\}$ represents the set-valued selection

$$COUNT(u.Depts.OfferCourses) > 500.$$

Both the directed path

$$(UNIVERSITY, DEPT, PROF)$$

and

$$(UNIVERSITY, DEPT, COURSE)$$

share common sub path $(UNIVERSITY, DEPT)$. It shows that the maximum common sub path expression $u.Depts$ between

$$u.Depts.FacultyMembers.Age$$

and

$$u.Depts.OfferCourses$$

has been factorized. The superedge

$$\{STUDENT, COURSE'', COURSE'\}$$

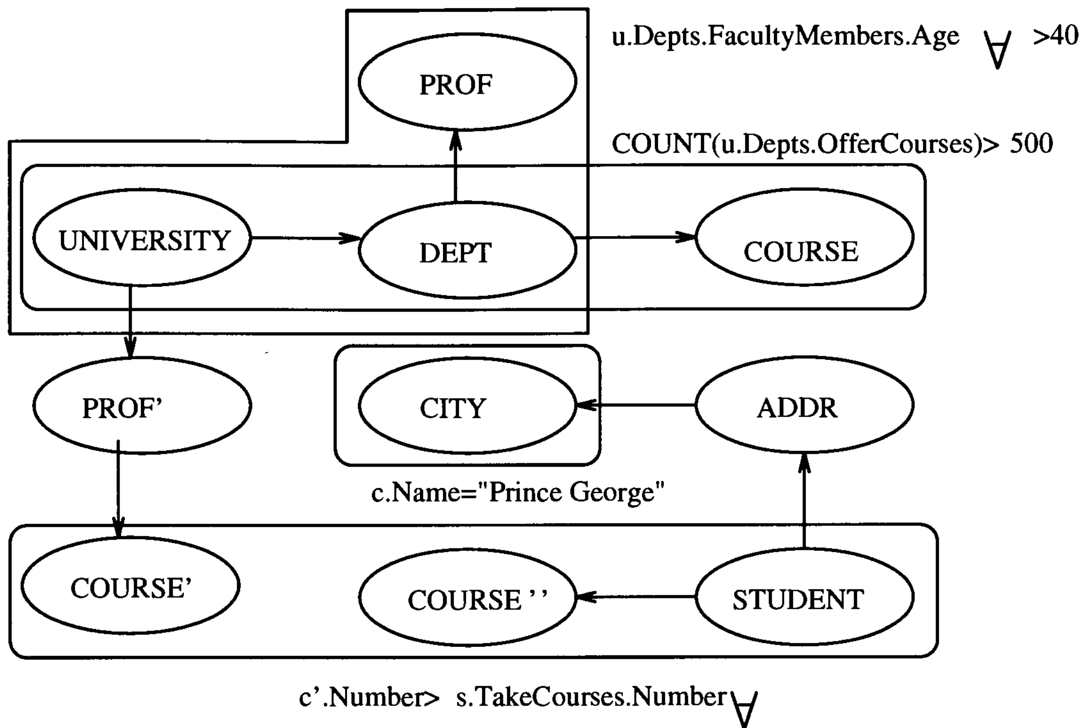


Figure 5.1: Query Graph of Example 5.10

denotes the set-valued join

$$c^{'}.Number > s.TakeCourses.Number_{\forall}$$

which is the result of the simplification and movement of the set-valued join

$$u.President.TeachCourses.Number_{\exists} > s.TakeCourses.Number_{\forall}.$$

Example 5.11 Find all the names of students who are from a metropolis or a large country with a population of over 20,000,000, and who are only taking computer

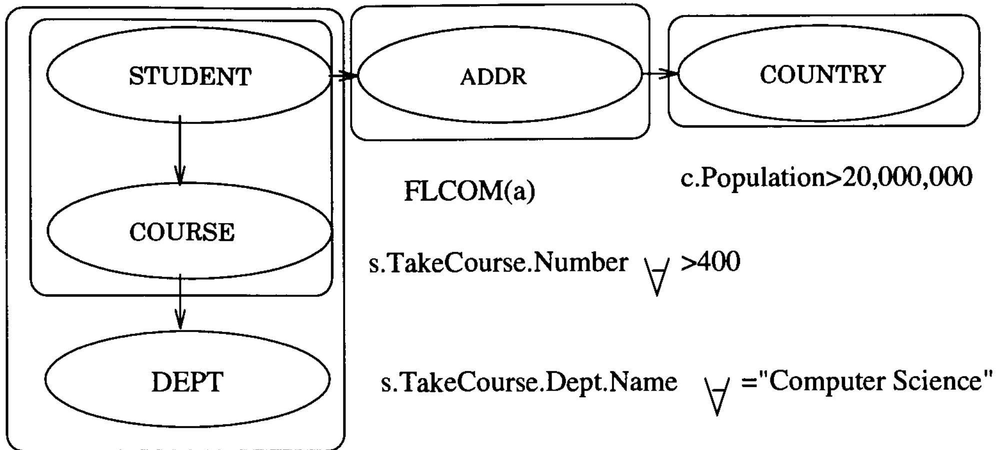


Figure 5.2: Query Graph of Example 5.11

science courses and taking courses over 400 level.

$$\begin{array}{ll} SELECT & s.Name \\ FROM & sSTUDENT \\ WHERE & FromLargeCountryOrMetro(s) \\ AND & s.HomeAddress.Country.Population > 20,000,000 \\ AND & s.TakeCourse.Dept.Name_{\forall} = \text{“}Computer\ \ Science\text{”} \\ AND & s.TakeCourse.Number_{\forall} > 400 \end{array}$$

□

Its query graph is in Figure 5.2. $\{STUDENT, COURSE\}$ is a superedge which denotes the set-valued selection:

$$s.TakeCourse.Number_{\forall} > 400.$$

The one node superedge $\{ADDR\}$, which denotes $FLCOM(a)$, is the result of the revelation of the method

$$FromLargeCountryOrMetro(s).$$

The one node superedge $\{COUNTRY\}$, which denotes

$$c.Population > 20,000,000,$$

represents the result of the simplification and movement of the single-valued selection:

$$s.HomeAddress.Country.Population > 20,000,000.$$

The superedge $\{STUDENT, COURSE, DEPT\}$, which denotes the set-valued selection:

$$s.TakeCourse.Dept.Name_{\forall} = \text{“}Computer\ \ Science\text{”}.$$

The directed paths

$$(STUDENT, COURSE)$$

and

$$(STUDENT, COURSE, DEPT)$$

represent the two path expressions

$$s.TakeCourse.Number$$

and

$$s.TakeCourse.Dept.Name$$

respectively. Their common sub path expression $s.TakeCourse$ has been factorized. The directed paths

$$(STUDENT, ADDR, COUNTRY)$$

and

$$(STUDENT, ADDR)$$

denote the path expression

$$s.HomeAddress.Country.Population$$

in the query and the maximal common sub argument path expression

$$s.HomeAddress$$

in the method $FromLargeCountryOrMetro(s)$ respectively. Their common sub path expression $s.HomeAddress$ has been factorized.

5.6.2 Query Plan Generation

Like in other query graph-based approaches, query plan generation corresponds to query graph manipulation. The query evaluation plans are generated according to the strategies proposed previously. Figure 5.3 illustrates the process of generating the query evaluation plan of Example 5.10. The query graph of Example 5.10 is in Figure 5.3(a) which is the same as in Figure 5.1 except that the labels of nodes have been changed for convenience.

1. Early single-valued selection

$$c.Name = \text{“}Prince\ \ George\text{”}.$$

   Remove the one-node superedges $\{a\}$ in Figure 5.3(a). The node remains but its label is changed into a'. The resulting graph is in Figure 5.3(b).

2. Single-valued joins

$$CITY \bowtie ADDR,\ \ ADDR \bowtie STUDENT,$$

$$STUDENT \bowtie COURSE'',\ \ PROF' \bowtie COURSE',$$

$$UNIVERSITY \bowtie PROF',\ \ DEPT \bowtie PROF,$$

$$DEPT \bowtie COURSE,\ \ UNIVERSITY \bowtie DEPT.$$

   The above joins correspond to the directed edges (9, a'), (7, 9), (7, 8), (5, 6), (1, 5), (2, 4), (2, 3) and (1, 2) respectively. Assume that the joins are performed in the above order. The resulting graphs are in Figure 5.3(c)-(j).

3. Set-valued selections and set-valued join

$$u.Depts.FacultyMembers.Age_{\forall} > 40,$$

$$COUNT(u.Depts.OfferCourses) > 500,$$

$$c'.Number > s.TakeCourses.Number_{\forall}.$$

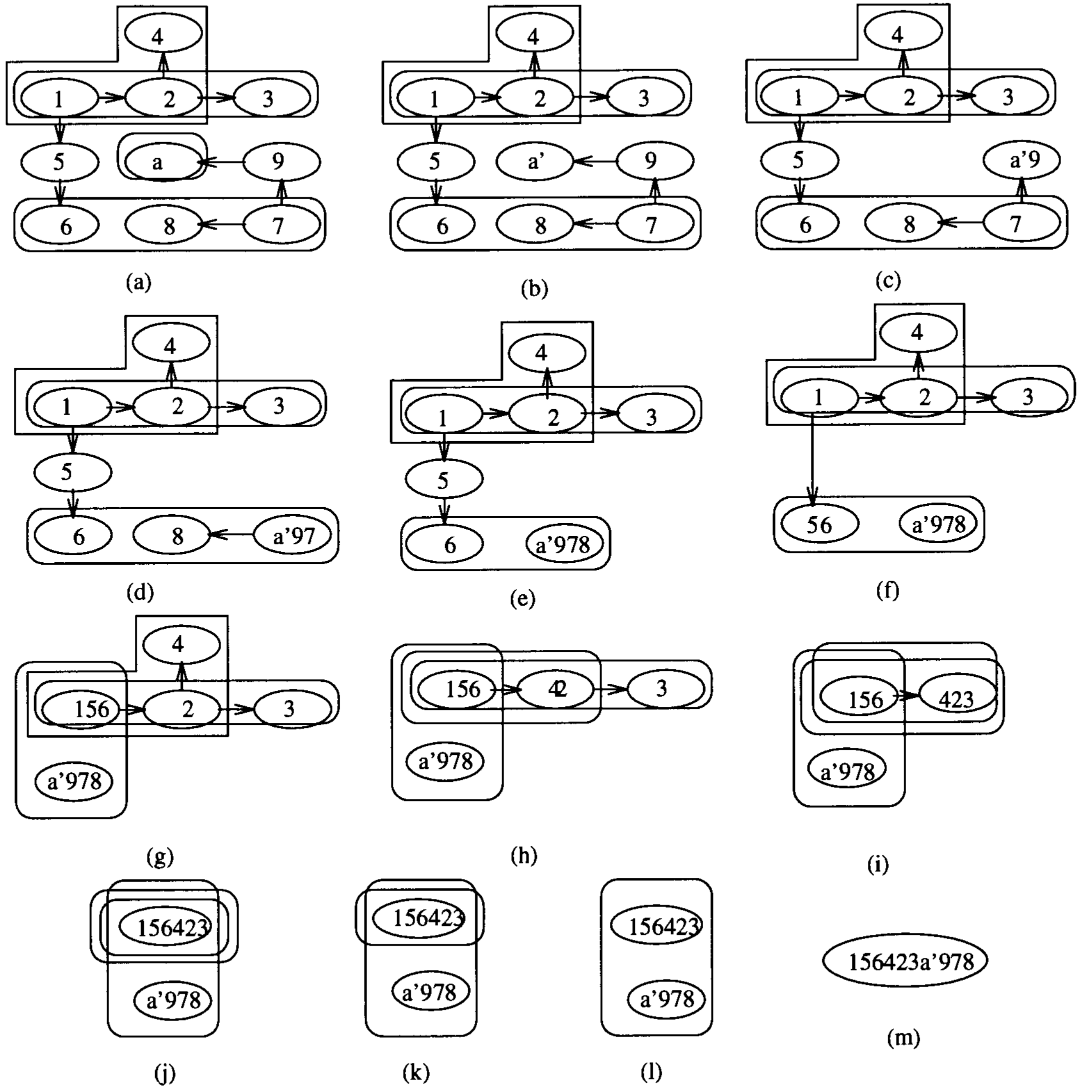


Figure 5.3: Query Plan Generation of Example 5.10

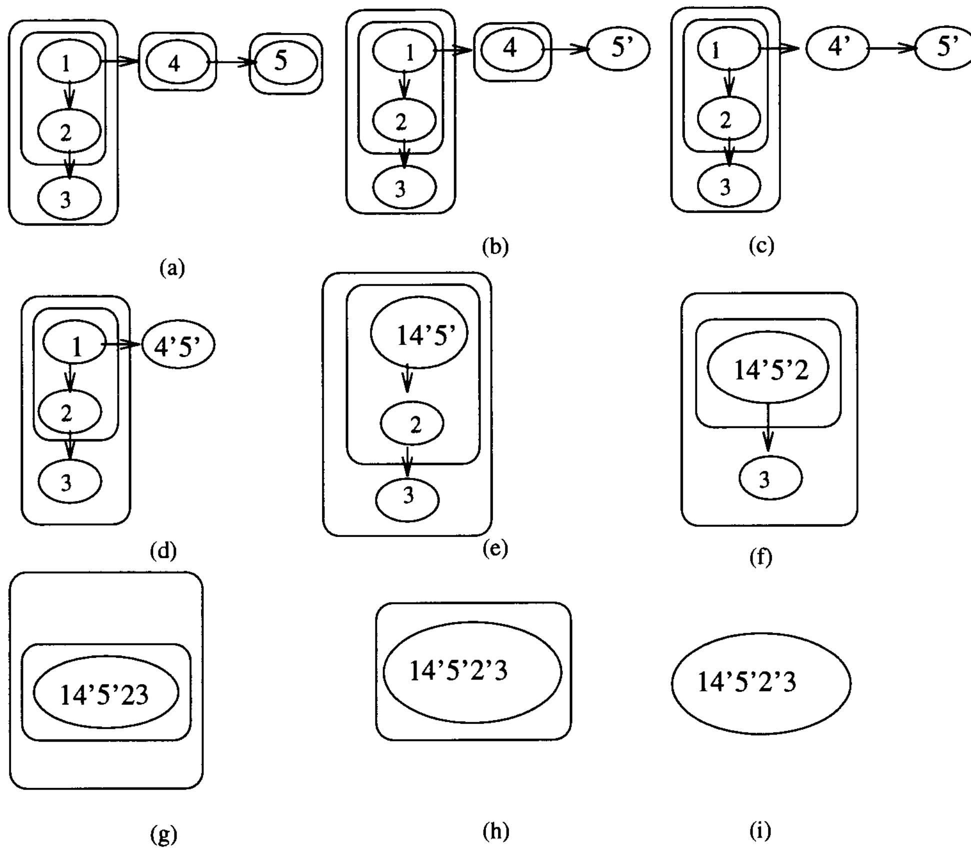


Figure 5.4: Query Plan Generation of Example 5.11

Remove the one-node superedge { 156423 } which corresponds to the above two set-valued selections in Figure 5.3(j). The resulting graphs are in Figure 5.3(k)-(l). Remove the superedge { 156423, a'978 } in Figure 5.3(l) which corresponds to the set-valued join $c'.Number > s.TakeCourses.Number_{\forall}$. The resulting graph is in Figure 5.3(m).

Figure 5.4 illustrates the process of generating query evaluation plan of Example 5.11. The query graph of Example 5.11 is in Figure 5.4(a) which is the same as in Figure 5.2 except that the labels of nodes have been changed for convenience.

1. Early single-valued selections include

$$c.Population > 20,000,000,$$

$$FLCOM(a).$$

Remove the corresponding two one-node superedges $\{5\}$, and $\{4\}$ in Figure 5.4(a). The two nodes 5 and 4 remain, however, their labels are changed into $5'$ and $4'$. The resulting graph is in Figure 5.4(b)(c).

2. Single-valued joins include

$$ADDR \bowtie COUNTRY, \quad STUDENT \bowtie ADDR,$$

$$STUDENT \bowtie COURSE, \quad COURSE \bowtie DEPT.$$

The above joins correspond to the directed edges $(4', 5)$, $(1, 4')$, $(1, 2)$ and $(2, 3)$ respectively. Assume that the joins are performed in the above order. The resulting graphs are in Figure 5.4(d)-(g).

3. Set-valued selections include

$$s.TakeCourse.Dept.Name_{\forall} = \text{"}Computer\ \ Science\text{"}$$

and

$$s.TakeCourse.Number_{\forall} > 400.$$

Remove the one-node superedge $\{14'5'23\}$ which corresponds to the above set-valued selections in Figure 5.4(g). The resulting graph is in Figure 5.4(h)(i).

5.7 Discussion

5.7.1 Optimization Strategies for Supporting Other Kinds of Navigations

Optimization strategies are presented for performing navigations with constraint conditions over class composition hierarchies, i.e., navigations via a sequence of attribute relationships. The class/subclass relationships are not explicitly expressed in path expressions since attributes of a superclass are inherited by its subclasses. For example, to find out the salary of a president of a university, a navigation is performed

from the class *UNIVERSITY* to the class *PRESIDENT* via the attribute *President*, the class/subclass relationship between the class *PRESIDENT* and the class *PROF*. The corresponding path expression is as follows

$$u.President.Salary.$$

The class/subclass relationship between the class *PRESIDENT* and the class *PROF* is not explicitly represented since the attribute *Salary* of the class *PROF* is inherited by the subclass *PRESIDENT*.

Similarly, the optimization strategies can be applied to navigations through a sequence of relationships specified by methods and/or rule deductions. For example, *Supervise* is a method of the class *PROF* which represents a relationship of supervising and supervised between a professor and a student. Students supervised by a professor p could be expressed as $p.Supervise$. Logically, the method *Supervise* is treated similarly as an attribute of *PROF*. A navigation, to find out professors who supervise some students from Canada, could be expressed as

$$p.Supervise.HomeAddress.Country.Name_{\exists} = \text{``}Canada\text{''},$$

which is a selection of type ($\exists$). However, there are some important differences between attributes and methods in the way they are implemented in database systems. Usually, attributes of an object are precomputed in databases and probably stored together with the object. For example, the object identifiers of the courses taught by a professor are stored together with the object professor. A navigation follows the object identifiers from the class *PROF* to the class *COURSE*. To find out the students supervised by some professors, however, a navigation is performed from the class *PROF* to the class *STUDENT* by invoking the method *Supervise*. A method may require intensive computation or many I/O on disks, therefore, a navigation via a relationship specified by methods and/or deduction rules could be more costly than that via an attribute relationship. Relationships specified by methods and/or deduction rules can also be precomputed. Instead of invoking methods and/or performing deductions, materialized results are accessed, for instance, to find out students supervised by a professor. The materialized results are, however, subject to database

updates. Extra mechanisms are required to maintain the precomputed relationships updated.

5.7.2 Method Materialization for Query Evaluation

Exploiting sharing of navigations for optimizing queries including user-defined methods are effective when there are large number of sharings between queries and methods. Precomputing user-defined methods or methods materialization is an alternative technique. Method materialization helps to avoid time-consuming computation, e.g., navigations, at run-time. It transforms object-at-a-time evaluation of methods into set-oriented retrieval of materialized results. However, there are two major issues related to method materialization:

- *Update maintenance.* A mechanism which bookkeeps all information used in computing a methods is needed to keep materialized method results updated. Two-levels of information are considered
  - *Schema level.* Schema information is used to capture global change which leads to recomputing all materialized results. For example, *RankAcademic* is a method which calculates the rank position of a student among his or her fellow students according to GPA. Any change in one student's GPA could lead to reordering of the rank positions of other students.
  - *Object level.* Object information is used to capture local change which only leads to recomputing one or some materialized results. For example, *AcademicStatus* is a method which evaluates a student's academic status such as excellent($GPA \geq 3.5$), good($2.5 \leq GPA < 3.5$), etc. The change of a student's GPA only leads to the change of this student's own academic status and does not change any other students' status.
- *Indexing and allocating materialized results.* Since the size of materialized results could be large, it is important that these results be stored in a way so that they can be efficiently retrieved during query execution.

- *Indexing on generic relationship.* There are some interesting relationships such as reflexive, symmetric, transitive and equivalent relationships. These relationships have some useful properties. For example, if a method corresponds to a symmetric relationship, the size of memories needed to store its materialized results can be reduced by half.

- *Employing class hierarchy information.* Class hierarchy information can be used to index materialized results for efficient retrieval. For example, *Friend* is a method such that if two persons p_1 and p_2 are friends then $Friend(p_1, p_2)$ returns *True*. Suppose that there is a class hierarchy rooted at *PERSON*. *STUDENT* and *PROF* are two subclasses of *PERSON*. *PhDStudent* and *MSStudent* are two subclasses of *STUDENT*. *ASSISTANTProf*, *ASSOCIATEProf* and *FullProf* are subclasses of *PROF*. Some queries may only inquire whether some PhD students and full professors are friends, other queries may only like to know whether some students are friends. Therefore class hierarchy information can be used to retrieve only the materialized results of *Friend* between *PhDStudent* and *FullProf* and between *STUDENT* and *STUDENT* which may be only a small part of the materialized results.

5.7.3 Integrating with Indexing Techniques

The proposed optimization techniques can be integrated with indexing methods such as join index hierarchies and nested indices. These indices are effective structures for supporting efficient associative searches and/or navigations. However, they are constructed for frequently performed navigations or associative searches because they might sacrifice disk space and require update maintenance. The proposed optimization techniques do not require extra space or update overhead, and accelerate navigations not supported by these index structures. For example, if there is a nested index on the path

$$c.Dept.Name$$

where c is an object in the class *COURSE*, the selection condition

$$p.Supervise.TakeCourse.Dept.Name_{\exists} = \text{“}Computer\ \ Science\text{''}$$

can be evaluated efficiently by searching the nested index to eliminate departments other than computer science and computing the implicit joins among *PROF*, *STUDENT* and the collection of computer science courses. Unlike the nested indices which only supports associative search, join index hierarchies provide more flexible and efficient index structures which support both forward and backward along a navigation path. If

$$s.TakeCourse.Dept$$

is a subpath of a path supported by a join index hierarchy, the selection condition

$$p.Supervise.TakeCourse.Dept.Chairperson.Name_{\exists} = \text{“}Jeff\ \ Ullman\text{''}$$

can still be evaluated efficiently by searching join index hierarchy rather than computing the implicit joins among *STUDENT*, *COURSE* and *DEPT*. Integrating the proposed optimization techniques with join index hierarchies does accelerate query evaluation.

Furthermore, Theorems 5.2 and 5.4 suggest that not only selections and joins with aggregation functions could be transformed into the equivalent and more efficient forms of the selections and joins without the aggregation functions but also selections and joins without aggregation functions could be replaced by the selections and joins with aggregation functions. It thus suggests that the values of MAX and MIN of path expressions, for instance, could be used as index values when evaluating selections and joins. For example, $MIN(s) > c$ is equivalent to $s_{\forall} > c$. Therefore, $MIN(s)$ can be stored as an index value for evaluating the selection $s_{\forall} > c$.

5.7.4 Integrating with Techniques for Searching Optimal Query Evaluation Plan

The costs and selectivities of selection and join conditions are essential for choosing optimal query evaluation plans. Unlike simple selection and join conditions in

relational database systems, the evaluation of selection and join conditions including user-defined methods may be costly. If the cost of a selection or join including a method is too high and the selectivity of the selection or join is not very sharp, the evaluation of the selection or join should be delayed even if it is a single-valued selection or join. Section 5.5.3 describes a simple approach to estimate the number of object references in a method. Other more complicated alternatives include sampling. This approach collects the cost and selectivity information by examining a small fraction of objects in databases.

It is possible, in principle, to determine the optimal order in which single-valued selections, single-valued joins, set-valued selections and set-valued joins are evaluated. In fact, set-valued selections and set-valued joins can be evaluated as soon as the implicit joins of ending classes of path expressions have been calculated. Their evaluations do not need to be delayed until after all those of single-valued selections and single-valued joins. However, if evaluations of set-valued selections and set-valued joins are delayed until after all single-valued selections and single-valued joins, the search strategy in Lanzellotte et al. [90] can be employed to determine optimal single-valued join orders in Step 2 of query evaluation plan generation in Section 5.6.2.

So far, we have considered conjunctive queries and their query graphs. Since all queries can be considered as unions of conjunctive queries, it is not hard to extend our work to handle non-conjunctive queries. A query can be represented by several query graphs each of which denotes a conjunctive query. These query graphs can be manipulated and the results can be combined into a whole query result.

5.8 Summary

In this chapter, the optimization of queries containing complex selections, joins, aggregations and encapsulated methods is studied. It has been clearly demonstrated that different types of selections and joins require different kinds of optimization strategies, and some set-valued selections and joins can be transformed into equivalent and more efficient forms of selections and joins. Sharing of navigation information between queries and methods can be effectively exploited for efficient query evaluation.

Optimization strategies have been proposed and incorporated to support efficient processing of complex queries.

Chapter 6

Recursive Query Evaluation

6.1 Introduction

Although recursive query evaluation has been investigated extensively in deductive database systems, it is not well understood whether and how the existing recursive query evaluation methods can be extended to handle DOOD recursive queries [114]. In deductive databases, data are conceptually grouped by properties. The information about one object, e.g., a student, may be spread in different relations which, in turn, are characterized by predicates. The deductive query evaluation methods center on the evaluation of variables and predicates and explore available constraints, e.g., query constraints and integrity constraints. The evaluations are performed either bottom-up from database facts, top-down from query goals, or the integration of the two approaches. In DOOD systems, however, data are grouped around objects described by syntactic notions such as object molecules. Therefore, it is natural to adapt the deductive recursive query evaluation methods and to focus on the evaluation of variables and object molecules.

Higher-order features complicate unifications [75]. Variables are allowed to appear in the places where class, attribute and method names do. Conventional most general unifiers may not exist and are replaced by the complete sets of most general unifiers [75]. Thus the algorithms for unifying two object molecules could be exponential since the number of alternative matches between the two object molecules could be

exponential. Furthermore, higher-order features complicates the exploration of the regularities of connections among variables since there could be more than one way to expand a rule if there were more than one most general unifier.

Example 6.1 Most general unifier. The following is a DOOD rule

$$U : C[X \to S; Y \to T] \quad \longleftarrow \quad V : C[X_1 \to S_1; Y_1 \to T_1], p_1(U, V), \ldots$$

where p_1 is a predicate. The complete set of the most general unifiers between

$$U : C[X \to S; Y \to T]$$

and

$$V : C[X_1 \to S_1; Y_1 \to T_1]$$

is

$$\{\{U\backslash V, X\backslash X_1, Y\backslash Y_1, S\backslash S_1, T\backslash T_1\}, \{U\backslash V, X\backslash Y_1, Y\backslash X_1, S\backslash T_1, T\backslash S_1\}\}.$$

None of the above two is more general than the other. This fact implies that there are two alternative ways to expand the rule. □

Constraints on higher-order variables may be more selective than those on conventional variables. For example, a constraint on a variable representing a class name would eliminate the number of classes to be considered during query evaluation, therefore, would effectively exclude the objects in those irrelevant classes from consideration. Similarly, a constraint on a variable denoting an attribute name could reduce the number of attributes accessed. The saving on navigation cost could be significant if nested attributes are involved. Obviously, query evaluation methods should capture query constraints, especially constraints on higher-order variables and apply the constraints at the early stage of query evaluations.

Although a DOOD program can be encoded into a first-order logic program [25, 75], the transformation may produce many predicates and rules. These predicates represent the access of attributes or the invocation of methods. Thus this process may change simple recursions into complex ones.

Example 6.2 Transforming a DOOD program into a datalog program. The following is a DOOD linearly recursive rule[1],

$$X : C[F \rightarrow U] \longleftarrow Y : C[F \rightarrow U_1; g \rightarrow U_1], \ \ p(X,Y,U,U_1).$$

where p is a predicate. A transformation similar to using *apply/n* [107] is to translate an attribute into a predicate. For example,

$$X : C[F \rightarrow U]$$

is translated into a predicate

$$Attr(C,X,F,U)$$

where the first argument represents the name of class, the second represents the object, the third represents the attribute name and the fourth represents the value of the attribute. The above rule is transformed into the following rule,

$$Attr(C,X,F,U) \longleftarrow Attr(C,Y,F,U_1), \ \ Attr(C,Y,g,U_1), \ \ p(X,Y,U,U_1).$$

which is a non-linearly recursive rule [48]. Obviously, a DOOD (single) linear recursion may be transformed into a non-linear recursion. Thus the transformation from a DOOD program to a datalog program may produce more complex recursions than the original ones in DOOD programs. □

In this chapter, we propose to extend the query-independent compilation and chain-based evaluation approach [53, 47, 46, 48] and to explore the regularities of connections among variables in object molecules.

- The query-independent compilation captures the bindings that could be difficult to be captured by other methods. The chain-based evaluation explores query constraints, integrity constraints, recursion structures, and other features of the programs with a set of interesting techniques, such as chain-following, chain-split and constraint pushing.

[1]See Definition 6.4.

- A normalization process is proposed to serve not only as a pre-processing stage for the compilation and evaluation but also as a tool for classifying recursions. A class of linear recursions, DOOD linear recursions, is identified which can be efficiently processed by the extension of the query-independent compilation and chain-based evaluation. The evaluation of nested linear recursions and the integration with other evaluation techniques are discussed as well.

Our proposal represents a promising approach toward efficient DOOD recursive query evaluation. In contrast to the existing systems which handle limited DOOD features, our method can deal with DOOD features including higher-order ones. Rather than translating programs into datalog, our method extends the query-independent compilation and chain-based evaluation to process DOOD recursions with normalization.

6.2 Normalization and Classification

A normalization process transforms a rule into a normalized form. Based upon normalization results, rules can be classified into different classes, e.g., non-recursive, recursive, single linearly and multiple linearly recursive rules. A class of linear recursions is identified which can be compiled into chain-forms and evaluated by chain-based evaluation techniques.

The normalization process includes *PullOut*, *Compose*, and *Associate* operations. Intuitively, a *PullOut* operation flattens a nested structure.

$$a[l_1 \rightarrow a_1; \ldots; l_i \rightarrow a_i[ll \rightarrow c]; \ldots; l_n \rightarrow a_n]$$
$$\stackrel{PullOut}{\Longrightarrow} a[l_1 \rightarrow a_1; \ldots; l_i \rightarrow a_i; \ldots; l_n \rightarrow a_n] \wedge a_i[ll \rightarrow c].$$

or

$$a[l_1 \rightarrow a_1; \ldots; l_i \rightarrow \{a_i[ll \rightarrow c]\}; \ldots; l_n \rightarrow a_n]$$
$$\stackrel{PullOut}{\Longrightarrow} a[l_1 \rightarrow a_1; \ldots; l_i \rightarrow \{a_i\}; \ldots; l_n \rightarrow a_n] \wedge a_i[ll \rightarrow c].$$

However, it is different from the unnest operation in that it "unnests" a nested structure by changing the syntactic expression of the nested structure without performing the actual unnest operation. This is made possible by the id-term a_i representing an object identity. The *Compose* operation collects all attributes and methods

regarding one object into the *same* object molecule.

$$a[l_1 \rightarrow a_1] \wedge \cdots \wedge a[l_n \rightarrow a_n] \stackrel{Compose}{\Longrightarrow} a[l_1 \rightarrow a_1; \ldots; l_n \rightarrow a_n].$$

The *Associate* operation makes explicit a class membership between an object and a class.

$$a : X \wedge a[l_1 \rightarrow a_1; \ldots; l_n \rightarrow a_n] \stackrel{Associate}{\Longrightarrow} a : X[l_1 \rightarrow a_1; \ldots; l_n \rightarrow a_n].$$

If $a : X$ is not present or there is more than one $a : X_i$, then choose X as the most specific class that a belongs to. If the root class[2] is the most specific class a belongs to, then the root class name is omitted.

After a normalization process, a rule consists of normalized object molecules in its rule head and rule body. Normalized object molecules are introduced to mimic the functionality of predicates in first-order logic.

Definition 6.1 Normalized object molecule. [3]

- A predicate object molecule is a normalized object molecule.
- is-a term P:Q or P::Q is a normalized object molecule where P and Q are variables or id-terms.
- Normalized data object molecules, denoted by

  $$C : D[l_1 \rightarrow C_1; ...; l_n \rightarrow C_n],$$

  where C is an id-term representing an object identity and D is an id-term denoting the name of a class which C belongs to. A label l_i, called the method expression, is of the form $Method@X_1, ..., X_m$[4] where $Method$ and X_i are id-terms representing a method name and an argument object respectively. C_i is an id-term or a set of id-terms representing the method result.

[2]Every object belongs to the root class.
[3]Without confusion, a normalized object molecule is simply called object molecule.
[4]Attribute expression *Attr* is a special case of method expression.

- Normalized signature object molecules, denoted by

$$C[l_1 \Rightarrow C_1; ...; l_n \Rightarrow C_n],$$

where C_i is an id-term representing a class name. A label l_i is of the form $Method@X_1, ..., X_m$ where $Method$ and X_i are id-terms representing a method name and an argument object type respectively. C_i is an id-term or a set of id-terms representing the type of the method result.

For example,

$$alex[HomeAddress \rightarrow addr[Country \rightarrow canada[Name \rightarrow \text{“}Canada\text{”}]]]$$

is not a normalized object molecule while

$$alex : PERSON[HomeAddress \rightarrow addr] \wedge addr : ADDR[Country \rightarrow canada]$$

$$\wedge canada : COUNTRY[Name \rightarrow \text{“}Canada\text{”}]$$

is a conjunction of normalized object molecules.

Definition 6.2 Normalized rules.

- All object molecules are normalized.
- All properties regarding same objects and their class memberships in the rule body are collected into the same normalized data object molecules.
- All signatures regarding same classes in the rule body are collected into the same normalized signature object molecules.

For example, the following is a normalized rule which defines a relationship between a student and a professor.

$$X : STUDENT[StudentOf@ \rightarrow \{Y\}] \longleftarrow$$

$$Y : PROF[TeachCourse \rightarrow \{C\}], X : STUDENT[TakeCourse \rightarrow \{C\}].$$

Definition 6.3 Recursive object molecule. An object molecule O_1 implies another object molecule O_2, say, $O_1 \Rightarrow O_2$, if there is a rule with an object molecule P_1 in the rule body and an object molecule P_2 as rule head such that P_1 can unify with O_1[5] and O_2 can be unified into P_2. If $O_1 \Rightarrow O_2$ and $O_2 \Rightarrow O_3$, then $O_1 \Rightarrow O_3$. If $O_1 \Rightarrow O_1$, then O_1 is called a recursive object molecule. If $O_1 \Rightarrow O_2$ and $O_2 \Rightarrow O_1$, then O_1 and O_2 are at the same deductive level. Otherwise, if $O_1 \Rightarrow O_2$ and $O_2 \not\Rightarrow O_1$, then O_1 is at lower deduction level than O_2.

In the previous example,

$$Y : PROF[TeachCourse \rightarrow \{C\}] \Rightarrow X : STUDENT[StudentOf@ \rightarrow \{Y\}]$$

and

$$X : STUDENT[TakeCourse \rightarrow \{C\}] \Rightarrow X : STUDENT[StudentOf@ \rightarrow \{Y\}].$$

Example 6.3 Database object browser.

$$\begin{array}{rcl} X[direct_ref@ \rightarrow \{Y\}] & \longleftarrow & X[U \rightarrow Y]. \\ X[direct_ref@ \rightarrow \{Y\}] & \longleftarrow & X[U \rightarrow \{Y\}]. \\ browser[ref@X \rightarrow \{Y\}] & \longleftarrow & X[direct_ref@ \rightarrow \{Y\}]. \\ browser[ref@X \rightarrow \{Y\}] & \longleftarrow & X[direct_ref@ \rightarrow \{Z\}], browser[ref@Z \rightarrow \{Y\}]. \end{array}$$

Since

$$browser[ref@X \rightarrow \{Y\}]$$

can be unified with

$$browser[ref@Z \rightarrow \{Y\}],$$

obviously,

$$browser[ref@X \rightarrow \{Y\}]$$

is a recursive object molecule.

Definition 6.4 Non-recursive and recursive rules.

[5]Either P_1 can be unified into O_1 or O_1 can be unified into P_1 (see Appendix D).

- *Non-recursive rule.* If all object molecules in the rule body are non-recursive, then the rule is called non-recursive rule.
- *Recursive rule.* If one or more of object molecules in the rule body are recursive, then the rule is called recursive rule.
- *Linearly recursive rule.* If only one of object molecules in the rule body can unify with the head object molecule and there exists a *unique most general unifier*, and all the other object molecules in the rule body are non-recursive then the rule is called linearly recursive rule.
- *Nested linearly recursive rule.* If only one of object molecules in the rule body can unify with the head object molecule and there exists a *unique most general unifier*, and all the other recursive object molecules in the rule body are at lower deduction level then the rule is called nested linearly recursive rule.
- *Multiple linearly recursive rule.* If only one of object molecules in the rule body can unify with the head object molecule and there exists more than one *most general unifier*, and all the other object molecules in the rule body are non-recursive then the rule is called multiple linearly recursive rule.

The condition that only one of object molecules in the rule body can unify with the head object molecule cannot guarantee a recursive rule is a linearly recursive one. In the following rule, for example,

$$U : C[X \to S; Y \to T] \longleftarrow V : C[X_1 \to S_1; Y_1 \to T_1], \cdots.$$

there are two most general unifiers such that

$$V[X_1 \to S_1; Y_1 \to T_1]$$

can be unified with

$$U[X \to S; Y \to T].$$

It implies that two different expansions can be generated when the rule is expanded. This example clearly shows that the correspondence between variables in

$$V[X_1 \rightarrow S_1; Y_1 \rightarrow T_1]$$

and

$$U[X \rightarrow S; Y \rightarrow T].$$

is established through the unification. Thus the multiple correspondences complicate the exploration of the regularities of connections among variables in object molecules.

Definition 6.5 Single linear recursion. A recursion is called a (single) linear recursion if it consists of only one linearly recursive rule and one or more other non-recursive rules.

The database object browser is a linear recursion. If the first two rules are changed into

$$X[direct_ref@ \rightarrow \{Y\}] \longleftarrow X[U \Rightarrow Y].$$
$$X[direct_ref@ \rightarrow \{Y\}] \longleftarrow X[U \Rightarrow \{Y\}].$$

then the recursion is a database schema browser, which is also a linear recursion.

6.3 Compilation and Evaluation of Linear Recursions

In datalog, the correspondence between the variables in a recursive rule head and a recursive predicate in the rule body is made explicitly by the argument positions in the predicates. In DOOD, however, this kind of relationship is established between the variables in the object molecule of the rule head and the recursive object molecule in the rule body through named attributes and methods. Since variables may appear in the places where attribute and method names do, the relationship can be determined by unification. In a linearly recursive rule, the most general unifier can be used to determine the unique correspondence between the variables in the rule head object molecule and the recursive object molecule in the rule body. This correspondence is the basis for expanding the linear recursion and capturing the regularities of connections among variables.

A DOOD linear recursion can be compiled into chain-forms, which make explicit the regularities of connections among variables and object molecules. A *chain* consists of a sequence of formulas with same structures. Every two consecutive chain elements share at least one variable. Each chain element is a sequence of connected non-recursive object molecules, called *a chain generating path.* Chain generating paths characterize the periodic property and the regularity of a DOOD linear recursion in expansion.

Theorem 6.1 A single linear recursion can be compiled into chain-forms.
Proof. See Appendix E.

Example 6.4 Compiling a linear recursion into chain form. The following is a linear recursion. The first rule is a linearly recursive rule while the second is the exit rule.

$$\begin{array}{lcl} X : c[f \rightarrow U, F@ \rightarrow V] & \longleftarrow & X_1 : c[f \rightarrow U_1, F@ \rightarrow V_1], p(U,U_1), q(V,V_1), r(X,X_1). \\ X : c[f \rightarrow U, F@ \rightarrow V] & \longleftarrow & e(X,U,V,F). \end{array}$$

The most general unifier which unifies the recursive object molecule in the rule body of the first rule

$$X_1 : c[f \rightarrow U_1, F@ \rightarrow V_1]$$

into the object molecule in the rule head

$$X : c[f \rightarrow U, F@ \rightarrow V]$$

is

$$\{X \backslash X_1, U \backslash U_1, V \backslash V_1\}.$$

The exit rule is considered the 0th expansion. The 1st expansion can be generated by the unification of the recursive object molecule in the first rule with the exit rule,

$$X : c[f \rightarrow U, F@ \rightarrow V] \longleftarrow e(X_1, U_1, V_1, F), p(U,U_1), q(V,V_1), r(X,X_1).$$

The 2nd expansion is to expand the first rule with the most general unifier and to unify the recursive object molecule in the expansion with the exit rule.

$$\begin{array}{lcl} X : c[f \rightarrow U, F@ \rightarrow V] & \longleftarrow & e(X_2, U_2, V_2, F), p(U,U_1), p(U_1,U_2), \\ & & q(V,V_1), q(V_1,V_2), r(X,X_1), r(X_1,X_2). \end{array}$$

Similarly, the ith expansion is as below,

$$\begin{aligned} X : c[f \rightarrow U, F@ \rightarrow V] \quad \longleftarrow \quad & e(X_i, U_i, V_i, F), \\ & p(U, U_1), \cdots, p(U_{i-1}, U_i), \\ & q(V, V_1), \cdots, q(V_{i-1}, V_i), \\ & r(X, X_1), \cdots, r(X_{i-1}, X_i). \end{aligned}$$

Obviously there are three chains,

$$p(U, U_1), \cdots, p(U_{i-1}, U_i), \cdots,$$

$$q(V, V_1), \cdots, q(V_{i-1}, V_i), \cdots,$$

$$r(X, X_1), \cdots, r(X_{i-1}, X_i), \cdots.$$

In the first chain, for example, the ith chain element is $p(U_{i-1}, U_i)$. Each two consecutive chain elements $p(U_{i-1}, U_i)$ and $p(U_i, U_{i+1})$ share a variable U_i. Each chain actually represents a transitive closure. Consequently,

$$X : c[f \rightarrow U, F@ \rightarrow V]$$

can be considered as the union of all the expansions, *i.e.*,

$$X : c[f \rightarrow U, F@ \rightarrow V] = \bigcup_{i=1}^{\infty}(e(X_i, U_i, V_i, F), X = X_0, U = U_0, V = V_0,$$

$$p^i(U_{i-1}, U_i), q^i(V_{i-1}, V_i), r^i(X_{i-1}, X_i))$$

where

$$p^i(U_{i-1}, U_i) = \begin{cases} True & \text{if } i = 0, \\ p(U_{i-1}, U_i), p^{i-1}(U_{i-2}, U_{i-1}) & \text{if } i > 0. \end{cases}$$

$$q^i(V_{i-1}, V_i) = \begin{cases} True & \text{if } i = 0, \\ q(V_{i-1}, V_i), q^{i-1}(V_{i-2}, V_{i-1}) & \text{if } i > 0. \end{cases}$$

and

$$r^i(X_{i-1}, X_i) = \begin{cases} True & \text{if } i = 0, \\ r(X_{i-1}, X_i), r^{i-1}(X_{i-2}, X_{i-1}) & \text{if } i > 0. \end{cases}$$

□

The chain-based approach includes the algorithms for testing finite evaluability and termination, and a set of evaluation techniques such as chain following, chain-split and constraint pushing [47, 46, 48]. Since some functional predicates and built-in predicates are defined on infinite domains, the number of answers to a query may be infinite. Sometimes, even though the result is finite, inappropriate evaluation methods may lead to infinite intermediate results. To insure proper evaluation, two issues should be considered: finite evaluability and termination. The former means that an evaluation is performed on finite inputs and generates finite intermediate results at each iteration. The latter guarantees that an evaluation generates all the answers and terminates after a finite number of iterations. A set of finiteness constraints is employed to check the finite evaluability. For example, $apply_cons(X, Y, Z)$ is a functional predicate corresponding to the list construction function *cons*. The finiteness constraint

$$Z \longrightarrow X, Y.$$

holds because a finite number of values for Z will determine the finite number of values for both X and Y. Similarly, a set of monotonic constraints is used to test the termination of an evaluation. For instance, list manipulations often result in shrinking or growing of lists. *cons* makes a list longer while *cdr* makes a list shorter.

If query constraints make all the object molecules in a chain generating path immediately finitely evaluable, a chain following evaluation can be performed. However, some object molecules in a chain generating path may not be immediately finitely evaluable with the currently available constraints, then the chain is not immediately finitely evaluable. The chain-split evaluation is performed by splitting the chain generating path into the two portions: the *immediately evaluable portion* and the *buffered portion*. The former consists of the immediately evaluable object molecules while the latter contains those not immediately evaluable object molecules. Iteration evaluation is performed on the immediately evaluable portion until no more answers are generated. A reverse iteration evaluation is conducted with the results from immediately evaluable portion and exit portion (exit rules). Query constraints can be pushed into chain-forms (iterative evaluation) to eliminate irrelevant data from further iterative evaluations.

Example 6.5 Append. A linear recursion, *append*, is defined by the following rules where *cons* is a list construction function and *list(T)* is a type of a parameterized list whose elements are of type T.

$$\begin{array}{rcl} nil[append(T)@L \rightarrow L] & \longleftarrow & L : list(T). \\ cons(X, L)[append(T)@M \rightarrow cons(X, N)] & \longleftarrow & L : list(T)[append(T)@M \rightarrow N], \\ & & X : T. \end{array}$$

The first is the exit rule while the second rule is a linearly recursive rule which defines *append*. After the normalization and rectification[6], they becomes

$$\begin{array}{rcl} L : Listv[A@M \rightarrow N] & \longleftarrow & L = nil, M = N, apply_append(T, A), \\ & & apply_list(T, Listv). \\ L : Listv[A@M \rightarrow N] & \longleftarrow & L_1 : Listv[A@M \rightarrow N_1], \\ & & apply_cons(X, L_1, L), apply_cons(X, N_1, N), \\ & & X : T, apply_list(T, Listv), apply_append(T, A). \end{array}$$

They can be transformed into the chain-form,

$$L : Listv[A@M \rightarrow N] =$$
$$\bigcup_{i=0}^{\infty}(L = L_0, M = N_i, N = N_0, L_i = nil,$$
$$apply_cons^i(X_i, L_i, L_{i-1}, N_i, N_{i-1})).$$

where

$$apply_cons^i(X_i, L_i, L_{i-1}, N_i, N_{i-1}) =$$

$$\begin{cases} True & \text{if } i = 0, \\ apply_cons^{i-1}(X_{i-1}, L_{i-1}, L_{i-2}, N_{i-1}, N_{i-2}), & \\ apply_cons(X_i, L_i, L_{i-1}), apply_cons(X_i, N_i, N_{i-1}), & \\ X_i : T, apply_list(T, Listv), apply_append(T, A) & \text{if } i > 0 . \end{cases}$$

Here the two functional predicates

$$apply_cons(X_i, L_i, L_{i-1}),$$

[6]The definition of rectification is in Appendix E

$$apply_cons(X_i, N_i, N_{i-1})$$

and the is-a object molecule

$$X_i : T,$$

are connected because they share the variable X_i. The two functional predicates

$$apply_list(T, Listv)$$

and

$$apply_append(T, A)$$

are connected with the above is-a object molecule since they share the variable T. When both L and N are instantiated, the iterative evaluation can be performed on the chain generating path

$$apply_list(T, Listv), apply_append(T, A),$$

$$apply_cons(X_i, L_i, L_{i-1}), apply_cons(X_i, N_i, N_{i-1}), X_i : T.$$

The first two functional predicates

$$apply_list(T, Listv), apply_append(T, A),$$

are common to all the chain elements, therefore, can be factorized. For example,

$$? - [p_1, p_2][append(C)@M \to [p_1, p_2, p_3]], C = PERSON$$

is a query regarding lists of persons. Here p_i are object identities representing persons. *Chain following* can be performed to evaluate the query. In a chain following evaluation from query end to exit end, the lengths of the second arguments in the following two functional predicates

$$apply_cons(X_i, L_i, L_{i-1}), apply_cons(X_i, N_i, N_{i-1}).$$

are getting shorter by one after each iteration. This guarantees that the evaluation will terminate in the finite number of iterations. The query constraint

$$C = PERSON$$

confines the elements of the list to persons only. It should be evaluated as soon as possible to exclude other types of elements in the list. Thus,

$$A = append(PERSON).$$

From the two common functional predicates,

$$T = PERSON, Listv = List(PERSON).$$

In the first iteration, the remaining chain generating path is

$$apply_cons(X_1, L_1, [p_1, p_2]), apply_cons(X_1, N_1, [p_1, p_2, p_3]), X_1 : PERSON.$$

It implies that

$$X_1 = p_1, \quad L_1 = [p_2], \quad N_1 = [p_2, p_3].$$

In the second iteration, the remaining chain generating path is

$$apply_cons(X_2, L_2, [p_2]), apply_cons(X_2, N_2, [p_2, p_3]), X_2 : PERSON.$$

It implies that

$$X_2 = p_2, \quad L_2 = nil, \quad N_2 = [p_3].$$

In the third iteration, the remaining chain generating path is

$$apply_cons(X_3, L_3, nil), apply_cons(X_3, N_3, [p_3]), X_3 : PERSON.$$

It implies that further iteration will not generate any answers to the query. Thus the answer to the query is

$$M = N_2 = [p_3].$$

However, if one of L and N is not instantiated, some part of a chain generating path may not be *immediately evaluable.* Consider the query,

$$? - [p_1, p_2][append(C)@[p_3] \rightarrow N], C = PERSON.$$

Again,

$$T = PERSON, Listv = List(PERSON).$$

can be derived from the query constraint

$$C = PERSON$$

and the two common functional predicates. In the first iteration, the remaining chain generating path is

$$apply_cons(X_1, L_1, [p_1, p_2]), apply_cons(X_1, N_1, N_0), X_1 : PERSON.$$

The first functional predicate is finitely evaluable which derives

$$X_1 = p_1, \ \ L_1 = [p_2].$$

However, the second functional predicate is not finitely evaluable with the only instantiation

$$X_1 = p_1.$$

The remaining chain generation path can be split into two parts

$$apply_cons(X_1, L_1, [p_1, p_2]), X_1 : PERSON$$

and

$$apply_cons(X_1, N_1, N_0).$$

The first, *immediately evaluable portion*, can be evaluated iteratively. However, the second part, *buffered portion*, will not be evaluated until the *exit portion* is evaluated. In the second iteration, the immediately evaluable portion is

$$apply_cons(X_2, L_2, [p_2]), X_2 : PERSON.$$

It derives

$$X_2 = p_2, \ \ L_2 = nil.$$

In the third iteration, the immediately evaluable portion is

$$apply_cons(X_3, L_3, nil), X_3 : PERSON.$$

It implies that further iteration will not produce any answers to the query. The buffered portions can be evaluated via inverse iterations. In the previous second iteration,

$$X_2 = p_2, \quad N_2 = M = [p_3] \text{ (from the exit portion).}$$

therefore the buffered portion is

$$apply_cons(p_2, [p_3], N_1).$$

It implies that

$$N_1 = [p_2, p_3].$$

In the previous first iteration, the buffered portion is

$$apply_cons(p_1, [p_2, p_3], N_0).$$

It implies that

$$N = N_0 = [p_1, p_2, p_3]$$

which is the answer to the query. □

Example 6.6 Travel plan. There are three alternatives for traveling, by air, sea and train. A customer may choose the combination of the three or only one of them. The following are is-a object molecules

$$\begin{array}{rcl} flight_timetable & :: & trans_timetable \\ cruise_timetable & :: & trans_timetable \\ train_timetable & :: & trans_timetable \end{array}$$

which describe that flight, cruise and train are means of transportation. The following are signature object molecules which specify the typing constraints on the classes *trans_timetable*, *flight_timetable*, *cruise_timetable* and *train_timetable*, and on

the signature of the method *plan(X)*.

$$
\begin{array}{rl}
trans_timetable & [departure \Rightarrow CITY; arrival \Rightarrow CITY; \\
& departure_time \Rightarrow TIME; arrival_time \Rightarrow TIME; \\
& fare@ \Rightarrow REAL] \\
flight_timetable & [flight_no \Rightarrow INT; airplane_maker \Rightarrow STRING; \\
& class \Rightarrow INT] \\
cruise_timetable & [cruise_name \Rightarrow STRING; class \Rightarrow INT] \\
train_timetable & [train_no \Rightarrow INT] \\
travel & [plan(X)@CITY, CITY, TIME, TIME, REAL \\
& \Rightarrow \{list(trans_timetable)\}]
\end{array}
$$

The following is a linear recursion which defines the method *plan(X)*.

$$
\begin{array}{ll}
travel & [plan(X)@Dep, Arr, Dep_Time, Arr_Time, Fare \rightarrow \{cons(T, nil)\}] \longleftarrow \\
& T : X[departure \rightarrow Dep; arrival \rightarrow Arr; departure_time \rightarrow Dep_Time; \\
& arrival_time \rightarrow Arr_Time; fare@ \rightarrow F], X :: trans_timetable.
\end{array}
$$

$$
\begin{array}{ll}
travel & [plan(X)@Dep, Arr, Dep_Time, Arr_Time, Fare \rightarrow \{cons(T, L)\}] \longleftarrow \\
& T : X[departure \rightarrow Dep; arrival \rightarrow Int_Arr; departure_time \rightarrow Dep_Time; \\
& arrival_time \rightarrow Int_Arr_Time; fare@ \rightarrow F_1], X :: trans_timetable, \\
& travel[plan(X)@Int_Arr, Arr, Int_Dep_Time, Arr_Time, F_2 \rightarrow \{L\}], \\
& F = F_1 + F_2.
\end{array}
$$

The rules can be normalized and rectified into

$$
\begin{array}{ll}
travel & [P@Dep, Arr, Dep_Time, Arr_Time, Fare \rightarrow \{L\}] \longleftarrow \\
& T : X[departure \rightarrow Dep; arrival \rightarrow Arr; departure_time \rightarrow Dep_Time; \\
& arrival_time \rightarrow Arr_Time; fare@ \rightarrow Fare], X :: trans_timetable, \\
& apply_plan(X, P), apply_cons(T, nil, L). \\
travel & [P@Dep, Arr, Dep_Time, Arr_Time, Fare \rightarrow \{L\}] \longleftarrow \\
& T : X[departure \rightarrow Dep; arrival \rightarrow Int_Arr; departure_time \rightarrow Dep_Time; \\
& arrival_time \rightarrow Int_Arr_Time; fare@ \rightarrow F_1], X :: trans_timetable, \\
& travel[P@Int_Arr, Arr, Int_Dep_Time, Arr_Time, F_2 \rightarrow \{L_1\}], \\
& sum(F_1, F_2, Fare), apply_plan(X, P), apply_cons(T, L_1, L).
\end{array}
$$

and transformed into the chain-form.

$$travel[P@Dep, Arr, Dep_Time, Arr_Time, Fare \rightarrow \{L\}] =$$

$$\bigcup_{i=1}^{\infty}(travel_plan^i(I_{i-1}, I_i, DT_i, AT_i, L_i, F_i, S_i)$$

$$I_0 = Dep, I_i = Arr, DT_0 = Dep_Time, AT_i = Arr_Time,$$

$$S_i = 0, S_0 = F, L_i = [], L = L_0)$$

where

$$travel_plan^i(I_{i-1}, I_i, DT_i, AT_i, L_i, F_i, S_i) =$$

$$\begin{cases} True & \text{if } i = 0, \\ T_i : X[departure \rightarrow I_{i-1}, arrival \rightarrow I_i, departure_time \rightarrow DT_i, & \\ arrival_time \rightarrow AT_i, fare@ \rightarrow F_i], sum(F_i, S_i, S_{i-1}), & \\ apply_cons(T_i, L_i, L_{i-1}), apply_plan(X, P), & \\ X :: trans_timetable, & \\ travel_plan^{i-1}(I_{i-2}, I_{i-1}, DT_{i-1}, AT_{i-1}, L_{i-1}, F_{i-1}, S_{i-1}) & \text{if } i > 0 \end{cases}$$

The following query

$$? - travel[plan(X)@Dep, Arr, Dep_Time, Arr_Time, Fare \rightarrow \{L\}],$$
$$Dep : CITY[name \rightarrow \text{“}Vancouver\text{”}], Arr : CITY[name \rightarrow \text{“}Toronto\text{”}],$$
$$Dep_Time \geq 8, Arr_Time \leq 22, Arr_Time \geq 20, Fare \geq 400, Fare \leq 800,$$
$$X = flight_timetable.$$

is to find only air travel plans which depart from Vancouver after 8 am. and arrive at Toronto between 8 pm. and 10 pm. The fare should be between \$400 and \$800.

The constraint on high-order variable X

$$X = flight_timetable$$

should be pushed into the iterative evaluation first so that only the air travel is considered. Since the query constraints at arrival end

$$Arr : CITY[name \rightarrow \text{“}Toronto\text{”}], Arr_Time \leq 22, Arr_Time \geq 20$$

are more selective. The evaluation should start at this end. The query constraints at arrival end can be pushed into the iterative evaluation. The remaining query constraints can be applied at the end of the iteration. However, it could be benefitial to apply these constraints as early as possible. The query constraint

$$Fare \leq 800$$

can be pushed into the evaluation to eliminate those plans with fares higher than \$800. This is based upon the monotonically increasing property of the function *sum*.

$$Dep_Time \geq 8$$

can be used to exclude those plans with departure time earlier than 8 am. because the integrity constraint

$$Int_Arr_Time < Int_Dep_Time$$

indicates that Dep_Time is a monotonic argument. If only air and train travel means are considered, then the query can be posed as

$$\begin{aligned}
&? - travel[plan(X)@Dep, Arr, Dep_Time, Arr_Time, Fare \rightarrow \{L\}],\\
&Dep : CITY[name \rightarrow \text{``}Vancouver\text{''}], Arr : CITY[name \rightarrow \text{``}Toronto\text{''}],\\
&Dep_Time \geq 8, Arr_Time \leq 22, Arr_Time \geq 20, Fare \geq 400, Fare \leq 800,\\
&(X = flight_timetable \vee X = train_timetable).
\end{aligned}$$

The same strategies can also be applied here. □

6.4 Discussion

6.4.1 Compilation and Evaluation of Nested Linear Recursions

In the previous section, it is shown that a (single) linear recursion can be compiled into chain-forms, and the chain-based evaluation can be applied to the compiled chain-forms. This section illustrates through an example that nested linear recursions can also be compiled into chain-forms and evaluated with the chain-based evaluation.

Definition 6.6 Nested linear recursion. A recursion is called a nested linear recursion if each recursive object molecule is defined by one linearly or nested linearly recursive rule and one or more non-recursive rules.

In a nested linearly recursive rule, an recursive object molecule which does not unify with the head object molecule is at a lower deduction level than the head object molecule. Therefore, it can be treated as a non-recursive object molecule during the compilation and evaluation.

Example 6.7 Joint things. This nested linear recursion describes a generic relationship among a group of persons. For example, a group of students take same one course, a group of researchers work on same projects, etc.

$$\begin{aligned} X[joint(M)@nil \rightarrow \{Z\}] &\longleftarrow X : PERSON[M \rightarrow \{Z\}]. \\ X[joint(M)@cons(Obj, Rest) \rightarrow \{Z\}] &\longleftarrow Obj : PERSON[M \rightarrow \{Z\}], \\ &\qquad \neg member(Obj, Rest), Obj \neq X, \\ &\qquad X : PERSON[joint(M)@Rest \rightarrow \{Z\}]. \end{aligned}$$

$$\begin{aligned} &member(X, cons(X, L)). \\ &Memebr(X, cons(Y, L)) \longleftarrow Memebr(X, L). \end{aligned}$$

The recursive predicate (object molecule) *member* is at a lower deduction level than the object molecule

$$X[joint(M)@cons(Obj, Rest) \rightarrow \{Z\}].$$

It is treated as a non-recursive object molecule during the compilation of the first two rules. The above rules can be normalized and rectified into

$$\begin{array}{rcl} X : PERSON[J@L \rightarrow \{Z\}] & \longleftarrow & X : PERSON[M \rightarrow \{Z\}], \\ & & L = [], apply_joint(M, J). \\ X : PERSON[J@L \rightarrow \{Z\}] & \longleftarrow & X : PERSON[J@Rest \rightarrow \{Z\}], \\ & & Obj : PERSON[M \rightarrow \{Z\}], \\ & & apply_joint(M, J), \\ & & apply_cons(Obj, Rest, L), \\ & & \neg member(Obj, Rest), Obj \neq X. \end{array}$$

$$\begin{array}{rcl} member(X, L) & \longleftarrow & apply_cons(X, N, L). \\ Memebr(X, L) & \longleftarrow & apply_cons(Y, N, L), Memebr(X, N). \end{array}$$

which can be transformed into the chain form,

$$X : PERSON[J@L \rightarrow \{Z\}] =$$

$$\bigcup_{i=0}^{\infty}(Rest_i = nil, L = Rest_0, OBJ^i(Obj_i, Rest_i, Rest_{i-1}))$$

and

$$member(X, L) = \bigcup_{i=0}^{\infty}(L = L_0, Y_i = X, apply_cons^i(Y_i, L_i, L_{i-1}))$$

where

$$OBJ^i(Obj_i, Rest_i, Rest_{i-1}) = \begin{cases} True & \text{if } i = 0, \\ OBJ^{i-1}(Obj_{i-1}, Rest_{i-1}, Rest_{i-2}), & \\ Obj_i : PERSON[M \rightarrow \{Z\}], & \\ Obj_i \neq X, \neg member(Obj_i, Rest_i), & \\ apply_cons(Obj_i, Rest_i, Rest_{i-1}), & \\ apply_joint(M, J) & \text{if } i > 0. \end{cases}$$

and

$$apply_cons^i(Y_i, L_i, L_{i-1}) = \begin{cases} True & \text{if } i = 0, \\ apply_cons(Y_i, L_i, L_{i-1}), & \\ apply_cons^{i-1}(Y_{i-1}, L_{i-1}, L_{i-2}) & \text{if } i > 0. \end{cases}$$

For example, the following query is to find what john and a group of people mary, joe and mark are doing together.

$$? - john[joint(M)@[mary, joe, mark] \rightarrow \{Z\}].$$

The chain following method can be applied to evaluate the query since all the object molecules in the chain generating path are immediately evaluable. The two arguments of *member* are bounded, therefore, only the existence checking [47, 46] is needed to evaluate *member*. However, the chain-split method should be performed to evaluate the following query

$$? - john[joint(TakeCourses)@L \rightarrow \{Z\}].$$

The immediately evaluable portion is

$$Obj : PERSON[M \rightarrow \{Z\}], apply_joint(M, J).$$

while the bufferd portion is

$$apply_cons(Obj, Rest, L), \neg member(Obj, Rest), Obj \neq X.$$

□

In [49], the query-independent compilation and chain-based evaluation are extended to handle some non-linear recursions, e.g., Tower of Hanoi and QuickSort.

6.4.2 Integrating with Indexing Techniques

The chain-based evaluation can be integrated with the indexing techniques for supporting efficient navigations. For example, if only Canadians are considered in the lists of Example 6.5, then the recursion can be redefined as

$$\begin{aligned}
nil[append(T)@L \rightarrow L] &\longleftarrow L : list(T), T :: PERSON. \\
cons(X, L)[append(T)@M \rightarrow cons(X, N)] &\longleftarrow L : list(T)[append(T)@M \rightarrow N], \\
&\qquad T :: PERSON, \\
&\qquad X : T[HomeAddress \rightarrow A], \\
&\qquad A : ADDR[Country \rightarrow C], \\
&\qquad C : COUNTRY[Name \rightarrow \\
&\qquad \text{``}Canada\text{''}].
\end{aligned}$$

The following query only considers Canadian students.

$$? - [p_1, p_2][append(T)@M \rightarrow [p_1, p_2, p_3]], T = STUDENT.$$

A join index hierarchy, which supports navigations between the class PERSON and the class COUNTRY or a nested index, which supports associative search on the path expression

$$X.HomeAddress.Country.Name,$$

surely accelerates the chain-based evaluation of the recursion. Instead of evaluating

$$X : T[HomeAddress \rightarrow A], A : ADDR[Country \rightarrow C],$$

$$C : COUNTRY[Name \rightarrow \text{``}Canada\text{''}]$$

in the iteration, our method can access the join index hierarchy or the nested index.

6.4.3 Exploring Typing Information

Although typing information can be exploited to eliminate some semantically meaningless results as a consequence of the introduction of higher-order features, semantic information is still necessary to guarantee correct answers. For example, the following is a typical definition of transitive closure. G represents an attribute name.

$$\begin{array}{lcl} X[transitive_closure(G)@ \rightarrow \{Y\}] & \longleftarrow & X[G \rightarrow Y]. \\ X[transitive_closure(G)@ \rightarrow \{Y\}] & \longleftarrow & Z[transitive_closure(G)@ \rightarrow \{Y\}], \\ & & X[G \rightarrow Z]. \end{array}$$

Based upon type checking, it can be concluded that X, Y and Z are of the same type. G should be an attribute relationship defined and ranged on the same class that X, Y and Z belong to. The type checking rules out alternatives which G can match otherwise. For instance, G cannot be the attribute names such as *TakeCourse*, *HomeAddress*, etc. However, typing information cannot guarantee to exclude all semantically meaningless results. For example, the relationship *co-author* between professors is not transitive. Although *John* co-authors with *Mary* and *Mary* co-authors with *Mark*, *John* may not co-author with *Mark*.

6.5 Summary

In this chapter, the query-independent compilation and chain-based evaluation are extended to process a class of DOOD linear recursions. Instead of transforming DOOD programs into Horn-like programs, the DOOD programs are preprocessed into normalized forms. The normalization process helps not only to compile and evaluate DOOD recursions but also to classify recursions. It is interesting to see how the other deductive evaluation methods can be extended to handle DOOD recursions.

Chapter 7

Conclusion and Future Research

7.1 Summary

Deductive and object-oriented database systems provide powerful modeling facilities and highly declarative languages, but require efficient query evaluation to achieve high performance in advanced applications. In this thesis, we have investigated the influence of DOOD data model and language on query evaluation. As a result, several important issues have been identified, including support for efficient navigation or "pointer-chasing", optimization of queries in the presence of complex selections, joins and aggregations, user-defined methods and encapsulation, and recursive query evaluation. The following research results constitute the major contributions of the thesis.

- Join index hierarchies for efficient navigations.
- Query optimization in the presence of complex selections, joins, aggregations and encapsulated methods.
- DOOD recursive query compilation and evaluation.

7.2 Future Research

In addition to the problems identified in this thesis, there are other interesting issues in DOOD query evaluation.

- *Index structures over class hierarchy.* Classes in DOOD form hierarchies. A query can be posed against objects in a class including and/or excluding objects of its subclasses. It is, therefore, a very important issue how an index can provide an efficient access structure for both cases.

- *Extensibility to support abstract data types and search strategies.* DOOD systems support abstract data types, and provide a dynamic environment in which users can define new database types, and access databases via arbitrarily-defined methods. Query processing must adapt to the ever-changing environment, provide new algorithms and techniques for new database types, handle different kinds of complex queries with different but effective strategies, and incorporate newly developed techniques and algorithms into the systems.

7.2.1 Indexing over Class Hierarchy

Since DOOD supports class/subclass hierarchies, the access scope of a query posed against a class may be the instances of only that class or the instances of all its subclasses. Thus, it is important that an index for DOOD could support efficient retrieval of a class including or excluding its subclasses.

Kim et al. [78] propose a class hierarchy tree, called CH-tree, which maintains only one index tree for all the classes in a class hierarchy. The performance shows that CH-tree performs better than that which supports one index for each class in a class hierarchy. The problem is that CH-tree does not support class/subclass relationships naturally. Retrieval of values in one class of a class hierarchy is treated the same as retrieval of values in a hierarchy of classes.

Low, Ooi and Lu [91] present an index tree, called H-tree. A H-tree is maintained for each class in a class hierarchy. These H-trees are nested according to the class/subclass relationship. A H-tree of a class in a class hierarchy is nested with

H-trees of its immediate superclasses in the class hierarchy. The index structure is constructed as a hierarchy of index trees. The index supports subclass and superclass relationships naturally and efficiently.

The disadvantage of the H-tree is that the index structure is too complicated and requires many physical pointers among H-trees. The retrieval cost increases as the number of classes in a hierarchy does. In the retrieval of objects in some classes in a hierarchy, H-tree outperforms CH-tree. However, in the retrieval of all objects in a hierarchy, CH-tree could perform better than H-tree. Kilger and Moerkotte [77] take advantage of both CH-tree and H-tree, and propose CG-tree. It maintains one tree, CG-tree, which groups objects according to their key values. However, the objects belonging to the same class in a hierarchy are clustered in same pages or pages linked together. The leaf pages of a CG-tree are organized into several doubly linked lists, each corresponding to a class in a hierarchy. Each record in the second level of a CG-tree contains references for each class in a hierarchy. The other records in higher levels are similar to those in a B^+-tree. The experiments show that if an application requires queries posed against several classes in a hierarchy, and the number of classes is large, CG-tree outperforms CH-tree and H-tree. Independently, Sreenath and Seshadri [113] present a similar index structure, hcC-tree which solves conflicting requirements for querying only one class and all classes in a class hierarchy.

7.2.2 Extensibility to Support Abstract Data Types and Search Strategies

DOOD systems allow users to define new data types and access objects via arbitrary-defined methods. Efficient query processing requires that a query optimizer provide the extensibility to handle the changing environment by incorporating multiple strategies and newly developed techniques.

EXODUS [40, 19], its successor Volcano [42], and Starburst [45, 54] are the typical representatives in achieving the extensibility of query optimizers [20]. In EXODUS and Volcano, a query optimizer generator is used to produce a query optimizer from

a rule-based specification of data model, query operators, access methods, costs, applicable transformation rules, and execution algorithms. The extensibility is achieved by regenerating an optimizer with modifications to the new input information. Starburst also takes a rule-based approach. However, grammar-like rules are used to generate database access operators from low-level operators. The construction of these database access operators provides extensibility in the sense that different ways of constructing these operators produce different methods of accessing databases.

Lanzelotte and Valduriez [89] propose a solution of the extensibility of the search strategy in a query optimizer. Search strategies are specified independently from search spaces. A search space is formulated as follows: an initial state constitutes relations and predicates from an input query; a state corresponds to a join node in a processing tree [90]; a goal state is a join node corresponding to the complete processing tree; and an action is an expand method. Search strategies are classified into different classes, enumerative and randomized which include iterative, simulated annealing and genetic searches, and arranged into a search class hierarchy. The extensibility of the search strategy in a query optimizer is achieved by the ability to add new search strategies into the search strategy class hierarchy. Mitchell, Dayal and Zdonik [96] present an extensible architecture for controlling an optimization process by providing multiple optimization control strategies and the ability to add new control strategies. The optimizer consists of a collection of optimization regions. Each region can transform a query according to a particular strategy, a set of transformation rules and a cost model. The optimizer coordinates the movement of a query among the regions. Kemper, Moerkotte and Peithner [70] propose a similar extensible architecture for searching optimal query evaluation plans step by step. The architecture is organized as a sequence of regions on a blackboard. A query is initially expressed in an internal form and put in the first region. A knowledge source between each successive pair of the regions transforms a query by moving it from a lower region to the next higher one. It can access information, such as database statistical information, database schema, etc., generate several alternatives and put them in the following region. An A^* based algorithm is employed to search optimal query evaluation plans. The extensibility is achieved by the ability to add new knowledge sources between

any successive pairs of regions and to add new regions.

7.3 Concluding Remarks

DOOD systems enable natural representations of logical relationships among complex objects and classes with attribute relationships, class/subclass relationships and relationships specified by user-defined methods and deduction rules. User-defined methods and deduction rules provide extensible mechanisms to define and model complex and ever-changing relationships among objects. Exploration of logical relationships among complex objects in user's queries, user-defined methods and deduction rules require performing efficient navigation operations to get over with "gotos' on disks". This thesis has presented promising approaches to the problems in DOOD query evaluation with the following features.

- This thesis promotes set-oriented evaluation of navigation operations by transforming object-at-a-time navigation operations into set-oriented access of appropriate join indices or constraint-based evaluation of a sequence of joins. Both forward and backward navigations among complex objects can be performed efficiently through a series of logical relationships.

- Join index hierarchies support efficient navigations via a sequence of logical relationships specified not only by attribute relationships and class/subclass relationships but also by user-defined methods and deduction rules. This is achieved by storing in join indices the precomputed relationships specified by user-defined methods and deduction rules. Thus, the evaluation of computation-intensive methods and deduction-intensive rules is transformed into efficient, set-oriented and associative access of the materialized relationships.

- "Push constraint inside navigation" is accomplished by identifying different types of constraint conditions and applying appropriate optimization strategies. Thus inexpensive but highly selective constraints can be evaluated to eliminate irrelevant objects before costly navigation operations are performed.

Common navigation operations are exploited among queries and user-defined methods by revealing the encapsulated methods. Query graphs are employed to represent different types of constraint conditions, to integrate different kinds of optimization strategies, and to generate efficient query evaluation plans.

- A normalization process is proposed to serve not only as a pre-processing stage for compilation and evaluation but also as a tool for classifying recursions. Rather than translating programs into datalog, our method extends the query-independent compilation and chain-based evaluation to process DOOD recursions with normalization. The query-independent compilation captures the bindings that could be difficult to be captured by other methods. The chain-based evaluation explores query constraints, integrity constraints, recursion structures, and other features of the programs with a set of interesting techniques, such as chain-following, chain-split, and constraint pushing.

We are planning to extend LogicBase [50, 51, 52] into a DOOD system and to incorporate the proposed strategies. This thesis will serve as a step towards efficient query evaluation in DOOD systems.

Appendix A

Evaluation of Some Parameters in Chapter 4

Table 4.1 lists some database parameters which are used in the analytical cost model in chapter 5. The probability of an object in C_{j-1} which does not reference a particular object in C_j is

$$\frac{\binom{|C_j|-1}{f_{j-1}}}{\binom{|C_j|}{f_{j-1}}}$$

i.e.,

$$1-\frac{f_{j-1}}{|C_j|}.$$

The probability of m objects in C_{j-1} which do not reference a particular objects in C_j is

$$(1-\frac{f_{j-1}}{|C_j|})^m.$$

The probability of a particular object in C_j which is referenced by m objects in C_{j-1} is

$$1-(1-\frac{f_{j-1}}{|C_j|})^m.$$

Therefore, the average number of objects in C_j which are referenced by these m objects in C_{j-1} is

$$|C_j| * (1 - (1 - \frac{f_{j-1}}{|C_j|})^m).$$

Hence,

$$fwd(i,j,k) = \begin{cases} \lceil p(|C_{i+1}|, f_i, k) \rceil & \text{if } j = i+1 \\ \lceil p(|C_j|, f_{j-1}, fwd(i, j-1, k)) \rceil & \text{if } j > i+1, \end{cases}$$

where

$$p(x,y,z) = x * (1 - (1 - \frac{y}{x})^z).$$

The number of tuples in $JI(i,j)$ is

$$|JI(i,j)| = |C_i| * fwd(i,j,1).$$

Similarly,

$$bwd(i,j,k) = \begin{cases} \lceil p(|C_i|, r_i, k) \rceil & \text{if } j = i+1 \\ \lceil p(|C_i|, r_i, bwd(i+1, j, k)) \rceil & \text{if } j > i+1 \end{cases}$$

The number of tuples in $JI(i,j)$ can also be calculated by

$$|JI(i,j)| = |C_j| * bwd(i,j,1).$$

Appendix B

Sample Database

Class PERSON	Class ADDR
SubClass STUDENT, PROF	Attributes
Attributes	Country:COUNTRY
Name: String	City: CITY
HomeAddress: ADDR	StreetName: String
Age: Int	StreetNumber: Int
	ZipCode: String

Class CITY	Class COUNTRY
Attributes	Attributes
Name: String	Name: String
Population: Int	Population: Int
Area: Real	Area: Real

```
Class DEPT
Attributes
Name: String
Chairperson: CHAIRPERSON
OfferCourses: Set of COURSE
OfferPrograms: Set of PROGRAM
FacultyMembers: Set of PROF

Class PROF
SubClass CHAIRPERSON, PRESIDENT
Attributes
Salary: Real
TeachCourses: Set of COURSE
Dept: DEPT
University: UNIVERSITY

Class CHAIRPERSON
Attributes
Dept: DEPT
University: UNIVERSITY

Class PROGRAM
Attributes
Name: String
Depts: Set of DEPT

Class STUDENT
Attributes
GPA: Real
TakeCourses: Set of COURSE
Major: PROGRAM

Class UNIVERSITY
Attributes
Name: String
President: PRESIDENT
Depts: Set of DEPT

Class COURSE
Attributes
Name: String
Number: Int
Dept: DEPT

Class PRESIDENT
Attributes
University: UNIVERSITY
```

```
FromLargeCountryAndMetro: STUDENT → Boolean
FromLargeCountryAndMetro(s:STUDENT):Boolean
  begin
    if (s.HomeAddress.Country.Population > 15,000,000
      and s.HomeAddress.Country.Area > 2,000,000)
      and (s.HomeAddress.City.Population > 1,000,000
```

```
    and s.HomeAddress.City.Area > 500)
  then return(True) else return(False)
end

FromLargeCountryOrMetro: STUDENT → Boolean
FromLargeCountryOrMetro(s:STUDENT):Boolean
begin
  if (s.HomeAddress.Country.Population > 15,000,000
    and s.HomeAddress.Country.Area > 2,000,000)
    or (s.HomeAddress.City.Population > 1,000,000
    and s.HomeAddress.City.Area > 500)
  then return(True) else return(False)
end

Top10WellPaidUniv: UNIVERSITY → Boolean
Top10PaidUniv(u:UNIVERSITY):Boolean
begin
  n=0;
  for each v ∈ UNIVERSITY do
    if (u.President.Salary ≤ v.President.Salary) then n=n+1;
  if (n ≥ 10) then return(False) else return(True);
end
```

Appendix C

Proof Sketch of Theorem 5.1 and Theorem 5.5

In the following, Theorem 5.1 is proved when $f = I$. The theorem can be proved similarly when $f \in \{MAX, MIN, COUNT, AVG, SUM\}$.

1. $f(o.A_1...A_n)_{q_1}\theta c_{q_2}$ is of type $\theta(\exists)$.

   Assume

   $$L = \{o | f(o.A_1...A_n)_{q_1}\theta c_{q_2},\ \ o \in O_0\}$$

   $$R = \pi_{(O_0)}(O_0 \bowtie ...O_{n-2} \bowtie \sigma_{(f(o_{n-1}.A_n)_{q_1}\theta c_{q_2})}(O_{n-1}))$$

   we now prove that $L = R$.

   Suppose $o_0 \in L$, then $f(o_0.A_1...A_n)_{q_1}\theta c_{q_2}$ is true. There exists an object path $(o_0, o_1, ..., o_{n-1})$ satisfying $o_0.A_1...A_n$ and $o_1 = o_0.A_1$, ..., $o_k \in o_{k-1}.A_k$, ..., $o_{n-1} = o_{n-2}.A_{n-1}$ if A_{n-1} is a single-valued attribute or $o_{n-1} \in o_{n-2}.A_{n-1}$ if A_{n-1} is a set-valued attribute, and $o_{n-1}.A_n\theta c_{q_2}$. Therefore, $(o_0, o_1, ..., o_n)$ is an instance of

   $$O_0 \bowtie ...O_{n-2} \bowtie \sigma_{(f(o_{n-1}.A_n)_{q_1}\theta c_{q_2})}(O_{n-1})$$

   and $o_0 \in R$.

   Suppose $o_0 \in R$, then there exists an object path $(o_0, o_1, ..., o_{n-1})$ which is an instance of

   $$O_0 \bowtie ...O_{n-2} \bowtie \sigma_{(f(o_{n-1}.A_n)_{q_1}\theta c_{q_2})}(O_{n-1})$$

$o_1 = o_0.A_1$, ..., $o_k \in o_{k-1}.A_k$, ..., $o_{n-1} = o_{n-2}.A_{n-1}$ when A_{n-1} is a single-valued attribute or $o_{n-1} \in o_{n-2}.A_{n-1}$ when A_{n-1} is a set-valued attribute, and $o_{n-1}.A_n\theta c_{q2}$. Therefore, $(o_0, o_1, ..., o_{n-1})$ satisfies $o.A_1...A_n$ and $f(o_0.A_1...A_n)_{q_1}\theta c_{q_2}$ is true, i.e., $o_0 \in L$. So $L = R$.

2. $f(o.A_1...A_n)_{q_1}\theta c_{q_2}$ is of type $\theta(\forall)$.

   Assume

$$L = \{o | f(o.A_1...A_n)_{q1}\theta c_{q_2},\ \ o \in O_0\}$$

$$R = \pi_{(O_0)}(O_0 \bowtie ...O_{k-2} \bowtie \sigma_{(f(o_{k-1}.A_k...A_n)_{q_1}\theta c_{q_2})}(O_{k-1} \bowtie ... \bowtie O_{n-1}))$$

   we now prove that $L = R$.

   Suppose $o_0 \in L$, then $f(o_0.A_1...A_n)_{q_1}\theta c_{q_2}$ is true. For all object path $(o_0, o_1, ..., o_{n-1})$ beginning with o_0 (actually o_0, ..., o_{k-1}) satisfying $o_0.A_1...A_n$, $o_1 = o_0.A_1$, ..., $o_k \in o_{k-1}.A_k$, ..., $o_{n-1} = o_{n-2}.A_{n-1}$ when A_{n-1} is a single-valued attribute or $o_{n-1} \in o_{n-2}.A_{n-1}$ when A_{n-1} is a set-valued attribute, and $o_{n-1}.A_n\theta c_{q_2}$. Therefore, all those $(o_0, o_1, ..., o_n)$ are instances of

$$O_0 \bowtie ...O_{k-2} \bowtie \sigma_{(f(o_{k-1}.A_k...A_n)_{q_1}\theta c_{q_2})}(O_{k-1} \bowtie ... \bowtie O_{n-1})$$

   and $o_0 \in R$.

   Suppose $o_0 \in R$. For all the instances $(o_0, o_1, ..., o_{n-1})$ of the following expression beginning with o_0 (actually o_0, ..., o_{k-1})

$$O_0 \bowtie ...O_{k-2} \bowtie \sigma_{(f(o_{k-1}.A_k...A_n)_{q_1}\theta c_{q_2})}(O_{k-1} \bowtie ... \bowtie O_{n-1})$$

   $o_1 = o_0.A_1$, ..., $o_k \in o_{k-1}.A_k$, ..., $o_{n-1} = o_{n-2}.A_{n-1}$ when A_{n-1} is a single-valued attribute or $o_{n-1} \in o_{n-2}.A_{n-1}$ when A_{n-1} is a set-valued attribute, and $o_{n-1}.A_n\theta c_{q_2}$. Therefore, all those instances $(o_0, o_1, ..., o_{n-1})$ satisfy $o.A_1...A_n$ and $f(o.A_1...A_n)_{q_1}\theta c_{q_2}$ is true, i.e., $o_0 \in L$. So $L = R$.

Theorem 5.5 is proved as follows.

1. $o.A_1...A_n$ is a single-valued path expression.

Since the maximal common sub argument path expression of the method m is a single-valued path expression, the method can be be rewritten as m' (see Example 5.9) such that

$$m(o) = m'(o.A_1...A_n).$$

m' can be considered as a method on the class O_n and

$$m(o) = m'(o_n).$$

where $o \in O_0$, $o_n \in O_n$ and $o_n = o.A_1...A_n$. Obviously, the first equation is true. Assume

$$L = \{o | m'(o.A_1...A_n) \;\; o \in O_0\}$$

and

$$R = \pi_{(O_0)}(O_0 \bowtie ... \bowtie \sigma_{m'(o_n)}(O_n)).$$

If $o \in L$, then $m'(o.A_1...A_n)$ is true. There exists an object path $(o, o_1, ..., o_n)$ satisfying $o.A_1...A_n$ such that $o_1 = o.A_1$, ..., $o_n = o_{n-1}.A_n$. Thus $m'(o_n)$ is true. $(o, o_1, ..., o_n)$ is an instance of the following expression,

$$O_0 \bowtie ... \bowtie \sigma_{m'(o_n)}(O_n)$$

Hence, $o \in R$.

Assume $o \in R$, then there exists an object path $(o, o_1, ..., o_n)$ which is an instance of the following expression,

$$O_0 \bowtie ... \bowtie \sigma_{m'(o_n)}(O_n).$$

Therefore $o_1 = o.A_1$, ..., $o_n = o_{n-1}.A_n$ and $m'(o_n)$ is true. Thus $(o, o_1, ..., o_n)$ satisfies $o.A_1...A_n$ and $m'(o.A_1...A_n)$ is true. Hence, $o \in L$. It is proved that

$$L = R.$$

2. $o.A_1...A_n$ is a set-valued path expression. The proof is similar to the above.

Appendix D

Unification Definition

The unification of id-terms, is-a object molecules and predicates is similar to that in classical first-order logic. However, the definition of unification for data object molecules and signature object molecules is different [75].

Definition D.1 Object molecule unification. A substitution σ is a *unifier* of object molecule $O_1 = I[\ldots]$ *into* object molecule $O_2 = I[\ldots]$ if and only if every attribute or method expression in $\sigma(O_1)$ is also an attribute or method expression in $\sigma(O_2)$.

The unification between two object molecules is asymmetric. For example, $I[M@X \rightarrow Y]$ can be unified into $I[M@U \rightarrow V, A \rightarrow W]$ with the substitution $\{M\backslash N, X\backslash U, Y\backslash V\}$, but not vice versa.

Definition D.2 Most general unifier. A unifier α is more general than a unifier β, denoted by $\alpha \prec \beta$, if and only if there is a substitution γ such that

$$\beta = \gamma \circ \alpha.$$

A unifier α is *most general* if for any unifier β, $\beta \prec \alpha$ implies $\alpha \prec \beta$.

There could be more than one most general unifiers such that one can be unified into another. For instance, both

$$\{M\backslash P, N\backslash Q, X\backslash U, Y\backslash V\}$$

and

$$\{M\backslash Q, N\backslash P, X\backslash V, Y\backslash U\}$$

are most general unifiers such that

$$I[M \rightarrow X; N \rightarrow Y]$$

can be unified into

$$I[P \rightarrow U; Q \rightarrow V].$$

A set Σ of most general unifiers of O_1 to O_2 is *complete* if for each unifier θ of O_1 to O_2 there is $\alpha \in \Sigma$ such that $\alpha \prec \theta$.

Appendix E

Proof Sketch of Theorem 6.1

In the following, we show that a single linear recursion can be compiled into chain-forms.

Proof Sketch.

1. *Correspondence between the method expressions in the rule head and the recursive object molecule in the rule body of a linearly recursive rule.* Since there exists a unique most general unifier such that the recursive object molecule in the rule body can unify with the rule head object molecule, there is a mapping from the method expressions of the recursive object molecule in the rule body to the method expressions of the rule head object molecule such that any two different method expressions of the recursive object molecule in the rule body are mapped to two different method expressions of the rule head object molecule.

2. *Rectify the linearly recursive rule.*

   - The rule head and the recursive object molecule are rectified by
     - performing function-predicate transformations [1],
     - replacing constants with variables and putting the corresponding equations between the constants and the variables in the rule body, and

[1] a function $f(X_1, ..., X_n)$ is replaced by fv and a functional predicate $apply_f(X_1, ..., X_n, fv)$ is added to the rule body.

  - making each variable occurrence only once and putting appropriate equations between the variables and new variables.

- Other object molecules are rectified by performing function-predicate transformations.

3. *Correspondence between variables in the rectified rule head and the rectified recursive object molecule.* Suppose

$$\{X_1, ..., X_m\}$$

are all the variables in the method expressions in the rule head which correspond to method expressions in the recursive object molecule in the rule body. According to 1), there should be m variables in the rectified recursive object molecule

$$\{Y_1, ..., Y_m\}$$

such that X_i corresponds to Y_i for $i = 1, ..., m$.

4. *V-matrix initialization and expansion* [53].

- *Identify U-connection.* Two object molecules in a rule body are *connected* if they share variable(s) with each other or with a set of connected object molecules. Two non-recursive object molecules in a rule body are *U-connected* if they share variable(s) with each other or with a set of U-connected object molecules. A set of variables are *U-connected* if they are in the same non-recursive object molecule or in the same set of U-connected non-recursive object molecules.

- *V-matrix initialization and expansion.* Copy

$$\{X_1, ..., X_m\}$$

into the first row of the V-matrix and

$$\{Y_1, ..., Y_m\}$$

into the second row. The rest of compilation process follows those in [53].

Bibliography

[1] R. Agrawal and N. H. Gehani. ODE (object database and environment): the language and the data model. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 36–45, Portland, OR, May 1989.

[2] H. Aït-Kaci and R. Nasr. LOGIN: a logic-programming language with built-in inheritance. *J. of Logic Programming*, 3:185–215, 1986.

[3] H. Aït-Kaci and A. Podelski. Towards a meaning of LIFE. *J. of Logic Programming*, 16(3, 4):195–234, 1993.

[4] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database systems manifesto. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Deductive and Object-Oriented Databases*, pages 223–240. Elsevier Science, 1990.

[5] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an object-oriented database system: the story of O2*. Morgan Kaufmann, San Mateo, CA,, 1992.

[6] F. Bancilhon and W. Kim. Object-oriented database systems: in transition. *ACM SIGMOD Records*, 19(4):49–53, Dec 1990.

[7] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic-sets and other strange ways to implement logic programs. In *Proc. Symp. Principles of Database Systems*, pages 1–15, Cambridge, MA, March 1986.

[8] J. Banerjee, W. Kim, and K. C. Kim. Queries in object-oriented databases. In *Proc. Int. Conf. Data Engineering*, pages 31–39, Los Angeles, CA, February 1988.

[9] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. GENESIS: an extensible database management system. *IEEE Trans. Software Engineering*, 14(11):1711–1730, Nov 1988.

[10] C. Beeri. Formal models of object oriented databases. In *Proc. 1st Int. Conf. Deductive and Object-Oriented Databases (DOOD'89)*, pages 405–429, Kyoto, Japan, Dec. 1989.

[11] C. Beeri and Y. Kornatzky. Algebraic optimization of object-oriented query languages. In *Proc. Int. Conf. on Database Theory*, pages 72–88, Paris, France, December 1990.

[12] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proc. Symp. Principles of Database Systems*, pages 269–283, San Diego, CA, March 1987.

[13] C. Beeri and R. Ramakrishnan. On the power of magic. *J. of Logic Programming*, 10(3,4):255–300, 1991.

[14] E. Bertino. An indexing technique for object-oriented databases. In *Proc. Int. Conf. Data Engineering*, pages 160–170, Kobe, Japan, April 1991.

[15] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Trans. Knowledge and Data Engineering*, 1(2):196–214, 1989.

[16] J. A. Blakeley, W. J. McKenna, and G. Graefe. Experiences building the open OODB query optimizer. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 287–296, Washington, DC, May 1993.

[17] J. Bocca. MegaLog: a platform for developing knowledge base management systems. In *Proc. 2nd Int. Symposium on Database Systems for Advanced Aplicaytions*, Tokyo, 1991.

[18] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, and H. Williams. The GemStone data management system. In W. Kim and F. H. Lochowsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 283–308. Addison-Wesley, Reading, MA, 1989.

[19] M. Carey, D. DeWitt, and S. L. Vandenberg. A data model and query language for EXODUS. In *Proc. 1988 ACM-SIGMOD Int. Conf. Management of Data*, pages 413–423, Chicago, IL, June 1988.

[20] M. Carey and L. Haas. Extensible database management systems. *ACM SIGMOD Records*, 19(4):54–60, Dec 1990.

[21] U. S. Chakravarthy, J. Grand, and J. Minker. Foundations of semantic query optimization for deductive databases. In J. Minker, editor, *Foundations of deductive databases and logic programming*, pages 243–274. Morgan Kaufmann, 1988.

[22] U. S. Chakravarthy, J. Grand, and J. Minker. Logic-based approach to semantic query optimization. *ACM Trans. Database Systems*, 13(2):162–207, June 1990.

[23] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *Proc. Int. Conf. Very Large Data Base*, pages 529–542, Dublin, Ireland, August 1993.

[24] S. S. Chawathe, M. S. Chen, and P. S. Yu. On index selection schemes for nested object hierarchies. In *Proc. Int. Conf. Very Large Data Base*, Santiago, Chile, September 1994.

[25] W. Chen, M. Kifer, and D. S. Warren. Hilog: A foundation for higher-order logic programming. *J. Logic Programming*, 15:187–230, 1993.

[26] W. Chen and D. S. Warren. C-logic for complex objects. In *Proc. Symp. Principles of Database Systems*, pages 369–378, 1989.

[27] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL system prototype. *IEEE Trans. Knowledge and Data Engineering*, 2:76–90, 1990.

[28] S. Choenni, E. Bertino, H. M. Blanken, and T. Chang. On the selection of optimal index configuration in OO databases. In *Proc. Int. Conf. Data Engineering*, pages 526–537, Phoenix, AZ, USA, February 1994.

[29] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 383–392, 1992.

[30] CODASYL. *CODASYL Data Base Task Group April 71 Report.* ACM, New York, NY, 1971.

[31] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 257–266, Washington, D. C., May 1993.

[32] S. Daniel, G. Graefe, T. Keller, D. Maier, D. Schmidt, and B. Vance. Query optimization in revelation, an overview. *IEEE Data Engineering Bulletin*, pages 58–62, June 1991.

[33] M. A. Derr, S. Morishita, and G. Phipps. The Glue-Neil deductive database system: design, implementation, and evaluation. *The VLDB Journal*, 3(2):123–160, April 1994.

[34] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Bemjamin/Cummings, 1989.

[35] D. H. Fishman, D. Beech, H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derret, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan. Iris: an object-oriented database management system. *ACM Trans. Office and Information Systems*, 5(1):48–69, Jan 1987.

[36] B. Freitag, H. Schutz, and G. Specht. LOLA: a logic language for deductive databases and its implementation. In *Proc. 2nd Int. Symposium on Database Systems for Advanced Applicaytions*, Tokyo, 1991.

[37] J. Frohn, G. Lausen, and H. Uphoff. Access to objects by path expressions and rules. In *Proc. Int. Conf. Very Large Data Base*, Santiago, Chile, September 1994.

[38] G. Graefe. Heap-filter merge join: A new algorithm for joining medium-size inputs. *IEEE Trans. Software Engineering*, 17(9), Sept 1991.

[39] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Survey*, 25(2):73–170, June 1993.

[40] G. Graefe and D. J. DeWitt. The exodus optimizer generator. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 160–172, 1987.

[41] G. Graefe and D. Maier. Query optimization in object-oriented database systems: a prospectus. In *Advances in Object-Oriented Database Systems*. Springer-Verlag, 1988.

[42] G. Graefe and W. J. McKenna. The Volcano optimizer generator: extensibility and efficient search. In *Proc. Int. Conf. Data Engineering*, pages 209–218, 1993.

[43] S. Greco, N. Leone, and P. Rullo. COMPLEX: an object-oriented logic programming system. *IEEE Trans. Knowledge and Data Engineering*, 4(4):344–359, August 1992.

[44] M. Guo, S. Y. W. Su, and H. Lam. An association algebra for processing object-oriented databases. In *Proc. Int. Conf. Data Engineering*, pages 154–162, 1991.

[45] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 377–388, 1989.

[46] J. Han. Chain-split evaluation in deductive databases. In *Proc. 8th Int. Conf. Data Engineering*, pages 376–384, Phoenix, AZ, Feb. 1992.

[47] J. Han. Compilation-based list processing in deductive databases. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Extending Database Technology - EDBT'92 [Lecture Notes in Computer Science 580]*, pages 104–119. Springer-Verlag, 1992.

[48] J. Han. Constraint-based query evaluation in deductive databases. *IEEE Trans. Knowledge and Data Engineering*, 6(1):96–107, January 1994.

[49] J. Han and L. V. S. Lakshmanan. Evaluation of regular nonlinear recursions by deductive database techniques. In *SFU CSS/LCCR Technical Report TR93-09*, Simon Fraser University, July 1993.

[50] J. Han, L. Liu, and Z. Xie. LogicBase: a system prototype for deductive query evaluation. In *Proc. ILPS Workshop on Programming with Logic Databases*, pages 146–160, Oct 1993.

[51] J. Han, L. Liu, and Z. Xie. Outline of LogicBase demonstration. In *Proc. ILPS Workshop on Programming with Logic Databases*, page 165, Oct 1993.

[52] J. Han, L. Liu, and Z. Xie. LogicBase: a deductive database system prototype. In *Proc. Int. Conf. Information and Knowledge Management*, pages 226–233, Nov 1994.

[53] J. Han and K. Zeng. Automatic generation of compiled forms for linear recursions. *Information Systems*, 17:299–322, 1992.

[54] L. M. Hass, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starbust mid-flight: As the dust clears. *IEEE Trans. Knowledge and Data Engineering*, 2(1):145–160, March 1990.

[55] J. M. Hellerstein. Practical predicate placement. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 325–335, Minneapolis, Minnesota, May 1994.

[56] J. M. Hellerstein and M. Stonebraker. Predicate migration: optimizing queries with expensive predicates. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 267–276, Washington, D. C., May 1993.

[57] K. A. Hua and C. Tripathy. Object skeletions: an efficient navigation structure for object-oriented database systems. In *Proc. Int. Conf. Data Engineering*, pages 508–517, Phoenix, Arizona, Feb 1994.

[58] IBM. *Information management system/virtual storage general information.* IBM Form Number GH20-1260, SH20-9025, SH20-9026, SH9027, 1978.

[59] Y. E. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 312–321, 1990.

[60] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: an analysis of strategy space and its implemetations for query optimization. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 168–177, 1991.

[61] Y. E. Ioannidis and E. Wong. Query optimization by simulated annealing. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 9–22, 1987.

[62] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Survey*, 16(2):111–152, Sept 1984.

[63] M. Jeusfeld and M. Staudt. Query optimization in deductive object bases. In J. C. Freytag, D. Maier, and G. Vossen, editors, *Query Processing for Advanced Database Systems*, pages 146–176. Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[64] Z. Jiao and P. M. D. Gray. Optimization of methods in a navigational query language. In *Proc. Int. Conf. Deductive and Object-Oriented Databases(DOOD)*, pages 22–42, 1991.

[65] K. Kato and T. Masuda. Persistent caching: an implementation technique for complex objects with object identity. *IEEE Trans. Software Engineering*, 18(7):631–645, July 1992.

[66] T. Keller, G. Graefe, and D. Maier. Efficient assembly of complex objects. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 148–157, Denver, CO, May 1991.

[67] A. Kemper, C. Kilger, and G. Moerkotte. Function materialization in object bases. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 258–267, 1991.

[68] A. Kemper and G. Moerkotte. Access support in object bases. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 364–374, Atlantic City, NJ, May 1990.

[69] A. Kemper and G Moerkotte. Advanced query processing in object bases using access support relations. In *Proc. Int. Conf. Very Large Data Base*, pages 290–301, Brisbane, Australia, August 1990.

[70] A. Kemper, G. Moerkotte, and K. Peithner. A blackboard architecture for query optimization in object bases. In *Proc. Int. Conf. Very Large Data Base*, pages 543–554, Dublin, Ireland, August 1993.

[71] A. Kemper, G Moerkotte, and M. Steinbrunn. Optimizing boolean expressions in object stores. In *Proc. Int. Conf. Very Large Data Base*, pages 79–90, Vancouver, Canada, August 1992.

[72] W. Kießling, H. Schmidt, W. Strauß, and G. Dünzinger. DECLARE and SDS: early efforts to commercialize deductive database technology. *The VLDB Journal*, 3(2):211–244, April 1994.

[73] M. Kifer, W. Kim, and Y. Sagiv. Quering object-oriented databases. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 393–402, San Diego, CA, May 1992.

[74] M. Kifer and G. Lausen. F-logic: a higher-order language for reasoning about objects, inheritance, and scheme. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 134–146, Portland, OR, May 1989.

[75] M. Kifer, G. Lausen, and J. Wu. Logical foundations for object-oriented and frame-based languages. In *Journal of ACM*, 1994.

[76] M. Kifer and J. Wu. A logic for object-oriented logic programming(Maier's o-logic revisted). In *Proc. Symp. Principles of Database Systems*, pages 379–393, March 1989.

[77] C. Kilger and G. Moerkotte. Indexing multiple sets. In *Proc. Int. Conf. Very Large Data Base*, Santiago, Chile, September 1994.

[78] K. C. Kim, W. Kim, and A. Dale. Indexing techniques for object-oriented databases. In W. Kim and F. H. Lochovsky, editors, *Object-oriented concepts, Databases, and Applications*, pages 371–394. Addison-Wesley, 1989.

[79] W. Kim. A new way to compute the product and join of relations. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 179–187, New York, NY, 1980.

[80] W. Kim. *Introduction to Object-Oriented Databases.* The MIT Press, 1990.

[81] W. Kim. Object-oriented databases: Definition and research directions. *IEEE Trans. Knowledge and Data Engineering*, 2:327–341, 1990.

[82] W. Kim. UniSQL/X unified relational and object-oriented database system. In *Proc. ACM-SIGMOD Conf. Management of Data*, page 481, Minneapolois, MN, May 1994.

[83] J. J. King. *Query optimization by semantic reasoning.* UMI Research Press, 1981.

[84] M. Kitsuregawa, H. Tanaka, and T. Motooka. Application of hash to data base machine and its architecture. In *Proc. 6th International Workshop on Database Machines*, June 1983.

[85] R. P. Kooi and Frankforth . Query optimization in Ingres. *IEEE Data Engineering Bulletin*, 5(3), Sept 1982.

[86] H. F. Korth and A. Silberschatz. *Database System Concepts, 2ed.* McGraw-Hill, 1991.

[87] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. On the logical foundations of schema integration and evolution in heterogeneous database systems. In *Proc. Int. Conf. Deductive and Object-Oriented Databases(DOOD)*, pages 81–100, Phoenix, AZ, USA, December 1993.

[88] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[89] R. S. G. Lanzelotte and P. Valduriez. Extending the search strategy in a query optimizer. In *Proc. Int. Conf. Very Large Data Base*, pages 363–373, Barcelona, Spain, September 1991.

[90] R. S. G. Lanzelotte, P. Valduriez, M. Ziane, and J. Cheiney. Optimization of nonrecursive queries in OODBs. In *Proc. Int. Conf. Deductive and Object-Oriented Databases(DOOD)*, Munich, Germany, December 1991.

[91] C. C. Low, B. C. Ooi, and H. Lu. H-tree: a dynamic associative search index for OODB. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 134–143, San Diego, CA, May 1992.

[92] W. Lu and J. Han. Distance-associated join indices for spatial range search. In *Proc. 8th Int. Conf. Data Engineering*, pages 284–292, Phoenix, AZ, Feb. 1992.

[93] D. Maier. A logic for objects. In *Proc. of Workshop on Foundations of Deductive Databases and Logic Programming*, pages 6–26, Washington, D. C., August 1986.

[94] D. Maier and J. Stein. Indexing in an object-oriented DBMS. In *Proc. IEEE Int. Workshop on Object-oriented Database System*, pages 171–182, Asilomar, Pacific Grove, CA, September 1986.

[95] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Survey*, 24(1):63–113, March 1992.

[96] G. Mitchell, U. Dayal, and S. B. Zdonik. Control of an extensible query optimizer: a planning -based approach. In *Proc. Int. Conf. Very Large Data Base*, pages 517–528, Dublin, Ireland, August 1993.

[97] I. S. Mumick, S. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *Proc. ACM-SIGMOD Conf. Management of Data*, May 1990.

[98] Objectivity. *Objectivity database system overview.* Objectivity, Inc., Menlo Park, CA, 1990.

[99] Ontos. *Ontos reference manual.* Ontos, Inc., Burlington, MA, 1993.

[100] J. Orenstein and F. Manola. Probe spatial data modelling and query processing in an image database application. *IEEE Trans. Software Engineering*, 14(5):611–629, May 1988.

[101] S. L. Osborn. Algebraic query optimization for an object algebra. Technical Report 251, University of Western Ontairo, 1989.

[102] M. Palmer and S. B. Zdonik. FIDO: a cache that learns to fetch. In *Proc. Int. Conf. Very Large Data Base*, pages 255–264, Barcelona, Spain, 1991.

[103] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. The CORAL deductive system. *The VLDB Journal*, 3(2):161–210, April 1994.

[104] R. Ramakrishnan and J. D. Ullman. A survey of research on deductive database system. *Journal of Logic Programming*, To appear.

[105] K. Ramamohanarao and J. Harland. An introduction to deductive database languages and systems. *The VLDB Journal*, 3(2):161–210, April 1994.

[106] D. Rotem. Spatial join indices. In *Proc. 7th Int. Conf. Data Engineering*, pages 500–509, Kobe, Japan, April 1991.

[107] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 442–453, Minneapolis, MN, May 1994.

[108] H.-J. Schek, H.-B. Paul, M. H. Scholl, and G. Weukum. The DASDBS project: Objectives, experiments, and future prospects. *IEEE Trans. Knowledge and Data Engineering*, 2:25–43, March 1990.

[109] P. G. Selinger, D. Astrahan, D. Astrahan, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 23–34, June 1979.

[110] G. M. Shaw and S. B. Zdonik. A query algebra for object-oriented databases. In *Proc. Int. Conf. Data Engineering*, pages 154–165, Los Angeles, CA, February 1990.

[111] E. J. Shekita and M. J. Carey. Performance enhancement through replication in an object-oriented DBMS. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 325–336, 1989.

[112] A. Silberschatz, M. Stonebraker, and J. D. Ullman. Database systems: achievements and opportunities. *Communications of the ACM*, 34(10):110–120, October 1991.

[113] B. Sreenath and S. Seshadri. The hcC-tree: an efficient index structure for object oriented databases. In *Proc. Int. Conf. Very Large Data Base*, pages 203–213, Santiago, Chile, September 1994.

[114] D. Srivastava, R. Ramakrishnan, P. Seshadri, and S. Sudarshan. Coral++: Adding object-orientation to a logic database language. In *Proc. Int. Conf. Very Large Data Base*, pages 158–170, Dublin, Ireland, August 1993.

[115] M. Stonebraker, R. Agrawal, U. Dayal, E. Neuhold, and A. Reuter. DBMS research at a crossroads: The vienna update. In *Proc. 19th Int. Conf. Very Large Data Bases*, pages 688–692, Dublin, Ireland, Aug. 1993.

[116] M. Stonebraker, L. Rowe, and M. Hirohama. The implementation of Postgres. *IEEE Trans. Knowledge and Data Engineering*, 2(1):125–142, Mar 1990.

[117] D. D. Straube and M. T. Ozsu. Queries and query processing in object-oriented database systems. *ACM Trans. Office and Information Systems*, 6(4):387–430, Oct 1990.

[118] A. Swami and A. Gupta. Optimizing large join queries. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 8–17, 1988.

[119] J. D. Ullman. *Principles of database and knowledge-base systems, Vol I, II.* Computer Science Press, 1989.

[120] J. D. Ullman. A comparison of deductive and object-oriented database systems. In C. Delobel et. al., editor, *Deductive and Object-Oriented Databases (DOOD'91) [Lecture Notes in Computer Science 566]*, pages 263–277. Springer Verlag, 1991.

[121] J.D. Ullman and C. Zaniolo. Deductive databases: achievements and future directions. *ACM SIGMOD Records*, 19(4):75–82, December 1990.

[122] J. Vaghani, K. Ramamohanarao, D. B. Kemp, Z. Somogyi, P. J. Stuckey, T. S. Leask, and J. Harland. The Aditi deductive database system. *The VLDB Journal*, 3(2):245–288, April 1994.

[123] P. Valduriez. Join indices. *ACM Trans. Database Systems*, 12(2):218–246, 1987.

[124] S. L. Vandenberg and D. J. DeWitt. Algebraic support for complex objects with arrays, identity, and inheritances. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 158–167, Denver, CO, May 1991.

[125] Versant. *Versant technical overview.* Versant Object Technologies, Inc., Menlo Park, CA, 1990.

[126] L. Vieille. From qsq towards qosaq: Global optimization of recursive queries. In *Proc. 2nd Int. Conf. Expert Database Systems*, pages 743–778, Vienna, VA, April 1988.

[127] L. Vieille, P. Bayer, V. Kuchenhoff, and A. Lefebvre. Eks-v1, a short overview. In *AAAI-90 Workshop on Knowledge Base Management Systems*, Boston, MA, July 1990.

[128] D. L. Wells, J. A. Blakeley, and C. W. Thompson. Architecture of an open object-oriented database management system. *IEEE Computer*, 25(10):74–82, October 1992.

[129] Z. Xie. Object query optimization containing encapsulated methods. In *Proc. Int. Conf. Information and Knowledge Management*, pages 451–460, Washington, DC, November 1993.

[130] Z. Xie and J. Han. Join index hierarchies for efficient navigation in object-oriented databases. In *Proc. Int. Conf. Very Large Data Base*, pages 522–533, Santiago, Chile, September 1994.

[131] Z. Xie and J. Han. Join index hierarchy: An indexing structure for efficient navigation in object-oriented databases. *submitted to IEEE Trans. Knowledge and Data Engineering*, 1994.

[132] Z. Xie and J. Han. Optimization of queries containing complex selections, joins and aggregations. *Journal of Computing and Information (Special Issue: Proceedings of the 6th International Conference on Computing and Information)*, 1(1):1410–1425, May 1994.

[133] S. B. Yao. Approximating block accesses in database organizations. *Communications of the ACM*, 20(4):260–261, April 1977.

[134] C. T. Yu and W. Sun. Automatic knowledge acquisition and maintenance for semantic query optimization. *IEEE Trans. Knowledge and Data Engineering*, 1(3):362–375, Sept 1989.