

การเข้าถึงลำดับชั้นรวมแบบต้นไม่สำหรับฐานข้อมูลเชิงวัตถุ



นาย พิชโยทัย มหัทธนาภิวัฒน์

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรดุษฎีบัณฑิต

สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย

ปีการศึกษา 2544

ISBN 974-03-0892-9

ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

ACCESS METHOD OF AGGREGATION HIERARCHY AS A TREE IN OODB

Mr. Pichayotai Mahatthanapiwat

A Dissertation Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy in Computer Engineering

Department of Computer Engineering

Faculty of Engineering

Chulalongkorn University

Academic Year 2001

ISBN 974-03-0892-9

Thesis Title Access Method of Aggregation Hierarchy as a Tree in OODB
By Mr. Pichayotai Mahatthanapiwat
Field of Study Computer Engineering
Thesis Advisor Associate Professor Wanchai Rivepiboon, Ph.D.

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial
Fulfillment of the Requirements for the Doctor 's Degree

..... Dean of Faculty of Engineering
(Professor Somsak Panyakeow, D.Eng.)

THESIS COMMITTEE

..... Chairman
(Associate Professor Somchai Prasitjutakul, Ph.D.)

..... Thesis Advisor
(Associate Professor Wanchai Rivepiboon, Ph.D.)

..... Member
(Associate Professor Supachai Tangwongsan, Ph.D.)

..... Member
(Assistant Professor Pornsiri Muenchaisri, Ph.D.)

..... Member
(Lecturer Twittie Senivongse, Ph.D.)

พิชโยทัย มัทธนาภิวัดณ์ : การเข้าถึงลำดับชั้นรวมแบบต้นไม้สำหรับฐานข้อมูลเชิงวัตถุ.
(ACCESS METHOD OF AGGREGATION HIERARCHY AS A TREE IN OODB)
อ.ที่ปรึกษา : รองศาสตราจารย์ ดร. วันชัย ใจไพบูลย์, 106 หน้า. ISBN 974-03-0892-9.

วิทยานิพนธ์นี้ได้เสนอวิธีการเข้าถึงข้อมูลเช่น วิธีการเข้าถึงเส้นทางปลายเสมือนโดยตรง, วิธีเพิ่มสัญลักษณ์แสดงตน และวิธีดรรชนีสาขา สำหรับการประมวลผลสอบถามกับลำดับชั้นรวมแบบต้นไม้สำหรับฐานข้อมูลเชิงวัตถุ และนำเสนออัลกอริทึมสำหรับกระบวนการสร้างสาขาจากคลาสที่มีรูปแบบลำดับชั้นรวมแบบต้นไม้ในฐานข้อมูล สำหรับข้อมูลของแต่ละสาขา จะมีการเก็บข้อมูลของวัตถุจากคลาสที่อยู่ในสาขานั้นและการเชื่อมโยงไปยังวัตถุอื่นของคลาสอื่นทำให้ไม่จำเป็นต้องใช้วิธีที่องคศาสตร์ในฐานข้อมูล การใช้ดรรชนีคุณลักษณะและดรรชนีรูปพรรณสำหรับสาขา จะช่วยให้กระบวนการสืบค้นทำได้เร็วขึ้น

นอกจากนี้ ยังได้มีการอธิบายถึงการดึงข้อมูลและการแก้ไขข้อมูลสำหรับวิธีการเข้าถึงแบบต่างๆสำหรับลำดับชั้นรวมแบบต้นไม้ และนำเสนอในสูตรของรูปแบบค่าใช้จ่ายในการจัดเก็บ, การดึงข้อมูล และการแก้ไขข้อมูล จากผลการวิเคราะห์ เมื่อมีการเปรียบเทียบรูปแบบค่าใช้จ่ายกับวิธีดรรชนีพจนานุกรมเส้นทางสำหรับหลายเส้นทาง พบว่าค่าใช้จ่ายในการจัดเก็บของดรรชนีสาขาจะน้อยกว่าวิธีดรรชนีพจนานุกรมเส้นทาง และค่าใช้จ่ายในการดึงข้อมูลเกือบทุกกรณีจะดีขึ้น แม้ว่าค่าใช้จ่ายในการแก้ไขข้อมูลสำหรับวิธีดรรชนีสาขาจะสูงกว่าของวิธีพจนานุกรมเส้นทางโดยส่วนใหญ่ แต่ค่าใช้จ่ายในการแก้ไขข้อมูลของวิธีดรรชนีสาขาจะดีกว่าวิธีพจนานุกรมเส้นทางถ้ามีการแก้ไขการเชื่อมโยงระหว่างต่างสาขา

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

ภาควิชา วิศวกรรมคอมพิวเตอร์ ลายมือชื่อนิสิต

สาขาวิชา วิศวกรรมคอมพิวเตอร์ ลายมือชื่ออาจารย์ที่ปรึกษา

ปีการศึกษา 2544 ลายมือชื่ออาจารย์ที่ปรึกษาร่วม

##4171818021 : MAJOR COMPUTER ENGINEERING

KEYWORD : ACCESS METHOD / QUERY PROCESSING / BRANCH INDEX / VIRTUAL
PATH / SIGNATURE / OBJECT-ORIENTED DATABASE

PICHAYOTAI MAHATTHANAPIWAT : ACCESS METHOD OF
AGGREGATION HIERARCHY AS A TREE IN OODB. THESIS ADVISOR :
ASSOC. PROF. WANCHAI RIVEPIBOON, Ph.D.106 pp.ISBN 974-03-0892-9.

This research proposed access methods such as the Direct Access to Terminal Virtual Path, the Virtual Path Signature and the Branch Index for query processing of the aggregation hierarchy as a tree in object-oriented databases. The algorithm of branch generation will be proposed to generate all branches for the tree aggregation of classes in the database. For each branch, the information of linking objects is stored so that class traversal methods can be eliminated. Using a set of attribute indexes and identity indexes for each branch, associative searching can be conveniently performed.

The discussion of the retrieval and update operation is performed among the access methods of aggregation hierarchy as a tree. Then, cost models in terms of storage overhead, retrieval cost and update cost are formulated. When compared with the Path Dictionary Index for multiple paths, the result shows that the Branch Index has less storage overhead and the retrieval cost is improving in most cases. Although most of the update cost of the Branch Index is higher than that of the Path Dictionary Index, It will be better than that of the Path Dictionary Index when the update is performed on the reference between different branches.

DepartmentComputer Engineering..... Student's signature.....

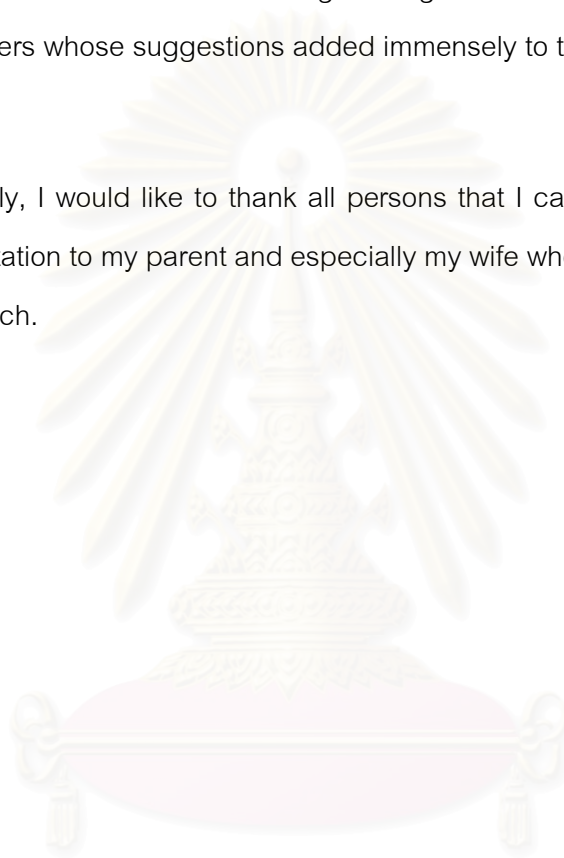
Field of studyComputer Engineering..... Advisor's signature.....

Academic year ...2001..... Co-advisor's signature.....

ACKNOWLEDGEMENTS

I would like to thank my advisor, Assoc. Prof. Wanchai Rivepiboon, Ph.D. for his useful advice and his help in doing the research. I am indebted to Assoc. Prof. Supachai Tangwongsan, Ph.D. for his teaching not only in the academic account but also in the lifestyle. I would like to thank all members of the committee for their suggestions and all members in Software Engineering Laboratory. I also owe a debt of thanks to the reviewers whose suggestions added immensely to the comprehensiveness of the research.

Finally, I would like to thank all persons that I cannot mention here and dedicate this dissertation to my parent and especially my wife who encourage me while I am doing the research.



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

TABLE OF CONTENTS

	PAGE
ABSTRACT IN THAI	iv
ABSTRACT IN ENGLISH.....	v
ACKNOWLEDGEMENTS.....	vi
TABLE OF CONTENTS.....	vii
TABLE OF TABLES.....	xi
TABLE OF FIGURES.....	xii
LIST OF NOTATION.....	xv
CHAPTER	
1. INTRODUCTION.....	1
1.1 Problem Statements.....	1
1.2 The Purpose of the Research.....	6
1.3 The Scope of the Research.....	6
1.4 The Benefits of the Research.....	6
1.5 The Method of the Research.....	7
2. RELATED WORKS.....	8
2.1 Theory.....	8
2.1.1 Object and Object Identifier.....	8
2.1.2 Attributes and Methods.....	9
2.1.3 Class.....	10
2.1.4 Class Hierarchy and Inheritance.....	10
2.1.5 Aggregation Hierarchy.....	10
2.1.6 Query Processing.....	11
2.2 Related Works.....	11
2.2.1 Indexing Techniques.....	12
2.2.1.1 Multi Index.....	12
2.2.1.2 Nested Index.....	12

TABLE OF CONTENTS (CONTINUE)

	PAGE
2.2.1.3 Path Index.....	13
2.2.1.4 Index of Direct Link.....	13
2.2.1.5 Path Dictionary Index.....	14
2.2.1.6 Other Researches.....	14
2.2.2 Signature Technique.....	16
3. DIRECT ACCESS TO TERMINAL VIRTUAL PATH.....	20
3.1 Organization.....	20
3.1.1 Terminal Virtual Path.....	20
3.1.2 Root-Terminal Virtual Path.....	22
3.1.3 Attribute Index.....	22
3.2 Database operation.....	23
3.2.1 Retrieval Operation.....	23
3.2.2 Update Operation.....	23
3.3 Cost Model.....	24
3.3.1 The Parameters of Cost Model.....	24
3.3.2 Storage Cost.....	25
3.3.3 Retrieval Cost.....	26
3.3.4 Update Cost.....	27
4. VIRTUAL PATH SIGNATURE.....	28
4.1 Organization.....	28
4.1.1 Terminal Virtual Path.....	28
4.1.2 Non-Terminal Virtual Path.....	29
4.1.3 Virtual Path Signature.....	29
4.2 Database Operation.....	31
4.2.1 Retrieval Operation.....	31
4.2.2 Update Operation.....	31
4.3 Cost Model.....	32

TABLE OF CONTENTS (CONTINUE)

	PAGE
4.3.1 The Parameters of Cost Model.....	32
4.3.2 Storage Cost.....	33
4.3.3 Retrieval Cost.....	33
4.3.4 Update Cost.....	34
5. BRANCH INDEX.....	36
5.1 Organization.....	36
5.1.1 Definitions.....	36
5.1.2 Algorithm of Branch Generation.....	38
5.1.3 Branch Index Organization.....	39
5.1.4 Details of the Branch Information.....	41
5.2 Implementation.....	43
5.3 Retrieval and Update Operation.....	47
5.3.1 Retrieval Operation.....	47
5.3.2 Update Operation.....	49
5.4 Cost Model.....	51
5.4.1 Storage Cost.....	52
5.4.2 Retrieval Cost.....	56
5.4.3 Update Cost.....	57
5.4.3.1 Update the Reference on the Same Branch.....	57
5.4.3.2 Update the Reference on Different Branches.....	59
6. COMPARISON OF ACCESS METHODS.....	60
6.1 Scope of the Comparison.....	60
6.1.1 The Number of Access Methods Used in Comparison.....	60
6.1.2 The Queries Used in Comparison.....	61
6.1.3 The Parameters Used in Comparison.....	62
6.2 Storage Cost.....	62
6.3 Retrieval Cost.....	69

TABLE OF CONTENTS (CONTINUE)

	PAGE
6.3.1 The Predicate Class is the Root Class.....	69
6.3.2 The Predicate Class is a Leaf Class.....	77
6.4 Update Cost.....	82
7. CONCLUSION AND PERSPECTIVE.....	90
7.1 Conclusion.....	90
7.2 Perspective.....	92
REFERENCES.....	93
APPENDICES.....	97
APPENDIX I Cost model of the Path Dictionary Index.....	98
APPENDIX II Letter of Acceptance from IJIT.....	103
APPENDIX III Abstract of Paper from IJIT.....	104
BIOGRAPHY.....	106



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

TABLE OF TABLES

TABLE	PAGE
Table 2.1 Signature generation example (in case $b = 16, k = 4$).....	17
Table 6.1 Parameters of cost models.....	62



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

TABLES OF FIGURES

FIGURE	PAGE
Figure 1.1 Aggregation hierarchy as a tree.....	3
Figure 2.1 Example of an object-oriented logical schema.....	9
Figure 3.1 Aggregation hierarchy as a tree.....	21
Figure 3.2 The structure of Terminal Virtual Path.....	21
Figure 3.3 The structure of root class-Terminal Virtual Path linking.....	22
Figure 4.1 The structure of the Terminal Virtual Path.....	28
Figure 4.2 The structure of the Non-Terminal Virtual Path.....	29
Figure 4.3 The structure of the Virtual Path Signature.....	30
Figure 4.4 The signature of the Terminal Virtual Path.....	30
Figure 5.1 Aggregation hierarchy as a tree.....	37
Figure 5.2 The Branch Index.....	41
Figure 5.3 An example of object instantiation.....	42
Figure 5.4 The structure of an entry of a branch.....	43
Figure 5.5 An example of the structure of an entry of the main branch.....	43
Figure 5.6 The structure of an information of a class in an entry of a branch.....	44
Figure 5.7 An example of the information for all classes of an entry of the main branch.....	45
Figure 5.8 An example of the information for all classes of an entry of the child branch.....	45
Figure 5.9 Non-leaf node record of the identity index and the attribute index.....	46
Figure 5.10 Leaf node record of the identity index and the attribute index.....	47
Figure 6.1 Aggregation hierarchy as a tree.....	60
Figure 6.2 The storage cost of access methods.....	68
Figure 6.3 The retrieval cost when the predicate class is the Person class and the target class is the Vehicle class.....	71
Figure 6.4 The retrieval cost when the predicate class is the Person class and the target class is the Company class.....	71
Figure 6.5 The retrieval cost when the predicate class is the Person class and the target class is the Course class.....	7

TABLE OF FIGURES (CONTINUE)

FIGURE	PAGE
Figure 6.6 The retrieval cost when the predicate class is the Person class and the target class is the Bank class.....	74
Figure 6.7 The retrieval cost when the predicate class is the Person class and the target class is the Engine class.....	74
Figure 6.8 The retrieval cost when the predicate class is the Person class and the target class is the University class.....	75
Figure 6.9 The retrieval cost when the predicate class is the Person class and the target class is the Computer class.....	75
Figure 6.10 The retrieval cost when the predicate class is the University class and the target class is the Vehicle class.....	77
Figure 6.11 The retrieval cost when the predicate class is the University class and the target class is the Company class.....	78
Figure 6.12 The retrieval cost when the predicate class is the University class and the target class is the Course class.....	78
Figure 6.13 The retrieval cost when the predicate class is the University class and the target class is the Bank class.....	80
Figure 6.14 The retrieval cost when the predicate class is the University class and the target class is the Engine class.....	80
Figure 6.15 The retrieval cost when the predicate class is the University class and the target class is the Computer class.....	81
Figure 6.16 The retrieval cost when the predicate class is the University class and the target class is the Person class.....	82
Figure 6.17 The update cost of reference between the Person class and the Vehicle class.....	85
Figure 6.18 The update cost of reference between the Vehicle class and the Company class.....	85
Figure 6.19 The update cost of reference between the Company class and the Bank class.....	86

TABLE OF FIGURES (CONTINUE)

FIGURE	PAGE
Figure 6.20 The update cost of reference between the Vehicle class and the Engine class.....	86
Figure 6.21 The update cost of reference between the Person class and the Course class.....	87
Figure 6.22 The update cost of reference between the Course class and the University class.....	87
Figure 6.23 The update cost of reference between the Person class and the Computer class.....	88



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

LIST OF NOTATIONS

- $N_{i,j}$: The number of objects in class j of branch i or path dictionary i .
- $A_{i,j}$: The complex attribute of class j of branch i or path dictionary i .
- $D_{i,j}$: Distinct value of complex attribute $A_{i,j}$.
- $UIDL$: The length of Object Identifier.
- P : Page size.
- pp : The size of page pointer.
- f : Average fan out from a non-leaf node.
- kl : Average length of a key value in attribute index.
- kv : Average key value for attribute index.
- SL : The length of start field in the branch information and s-expression.
- $OFFL$: The length of offset field in path dictionary.
- PL : The length of pointer in branch information.
- $SA_{i,j,k}$: Simple attribute k of class j on branch i .
- $U_{i,j,k}$: The number of distinct value for simple attribute.
- $q_{i,j,k}$: The ratio of shared attribute value.
- Pk_i : Reference sharing of the parent class of class i .
- $k_{i,j}$: Reference sharing of class j on branch i .
- $nlb_{i,j}$: The number of leaf branches of class j on branch i .
- LP : Leaf page of the identity index or attribute index.
- NLP : Non-leaf page of the identity index or attribute index.
- XP : Leaf node record of the attribute index.
- XI : Leaf node record of the identity index.
- h_{iden} : Height of the identity index.
- h_{attr} : Height of the attribute index.
- nLC : The number of leaf classes of the aggregation hierarchy.
- $nNLC$: The number of non-leaf classes of the aggregation hierarchy.
- ll : The length of link number.

LIST OF NOTATIONS (CONTINUE)

- cl : The length of link counter.
- S : The size of a signature.
- SZ : The size of an object.
- E : The average size of an entry in the signature file.
- K_S : The average size of a signature file.
- SE_{B_i} : The entry size of branch B_i .
- SS_{P_i} : The size of s-expression of path dictionary P_i .
- R : The average matching rate of a query signature.
- H_i : The number of ancestor classes from the root class to i^{th} class.
- N_{par} : The average number of parent objects for an object.
- N_p : The number of objects in the predicate class.
- $DTVP$: The Direct Access to Terminal Virtual Path.
- VPS : The Virtual Path Signature.
- BI : The Branch Index.
- PDI : The Path Dictionary Index.
- SC : The storage cost.
- RC : The retrieval cost.
- UC : The update cost.

CHAPTER 1

INTRODUCTION

The first chapter states the problem statements, which describe the motivation of this research, the purposes, scope and benefits of the research. Finally, the last section explains the method of this research.

1.1 Problem Statements

At present, object-oriented databases have been widely used in most engineering applications, such as Computer Aided Design (CAD), Computer Aided Manufacturing (CAM) and Geographical Information System (GIS). The complexity of data in these applications makes the conventional database, such as the relational database cumbersome to manage them. One of the benefits of the object-oriented database is from its data model [19]. In the object data model, the value of an attribute does not limit to a primitive value, such as integer, real or string, but the value of an attribute can be either a primitive value or a complex value. The complex value of an attribute is a unique Object Identifier (OID) of an object in a class [19]. If class C consists of an attribute A whose domain is class C' , class C can reference class C' from the attribute A . We call this relation of classes as an aggregation hierarchy. In the same way, class C' consists of an attribute A' whose domain is class C'' so that class C' can link to class C'' directly and class C can link to class C'' indirectly. If class N is referenced by class C either directly or indirectly and class N does not reference any classes, class N is called a leaf class of the aggregation hierarchy. On the other hand, class C is called the root class of the aggregation hierarchy if it references other classes, but it is not referenced by any classes. Any classes in the aggregation hierarchy that are between the root class and the leaf class are called intermediate classes. Class traversal methods for an aggregation hierarchy can be performed as forward traversal and reverse traversal. In the forward traversal approach, we start from one class and traverse to its child class by using the value of the complex attribute.

On the other hand, the reverse traversal approach traverses up to the parent classes. Usually, the forward traversal approach can perform well especially when the selection operation is performed on the start of the path expression [37] and by using the inherent pointer of the complex attributes. However, the reverse traversal approach has more trouble unless reverse pointers are implemented between classes. When there is a query, the class that the predicate is involved is called the predicate class and the class of the target objects is called the target class.

If the predicate class and the target class are far apart, i.e. there are several intermediate classes between the target class and the predicate class, cost of traversal will be high because of intermediate classes traversal. Therefore, many researches have been performed to reduce cost of class traversal whereas the associative searching is also in consideration. The indexing techniques are considered to accelerate database operations by constructing efficient access structures on a database given a certain physical implementation of the database. Secondary index on an attribute or a combination of attributes is useful for evaluating queries on a nested class in an object-oriented database. A classic research on index [6] has been done on an aggregation hierarchy, for example, multi index, nested index, path index. A join index hierarchy method [14] and [38] is proposed by extending the join index structure studied in relational databases. Other researches [5], [9], [15], [32], [33] on the aggregation hierarchy attempted to improve the performance of searching by using the concept form [6]. The researches on indexing technique of the inheritance hierarchy have been proposed in [18], [27] and [35]. Indexing techniques on both aggregation hierarchy and inheritance hierarchy are proposed by [3], [10], [12] and [13].

Most indexing techniques that are used for the aggregation hierarchy are proposed as a path scheme. However, for the application that a class schema is more complicated than a path, such as a tree, a new access method should be considered to cope with all classes in the aggregation hierarchy. An example of the aggregation hierarchy that forms a tree of linking classes is shown in Figure 1.1. It consists of eight classes, *Person*, *Vehicle*, *Company*, *Bank Engine*, *Course*, *University* and *Computer*.

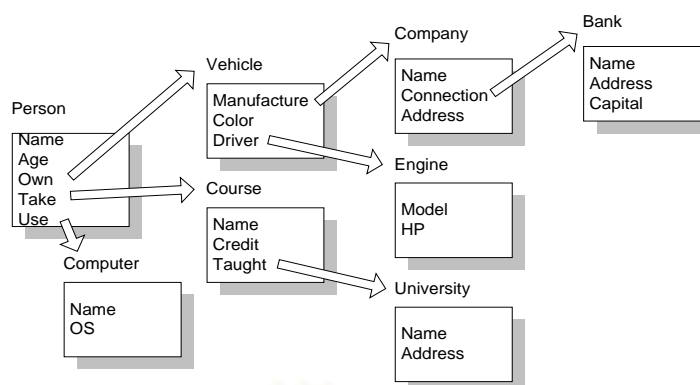


Figure 1.1 Aggregation hierarchy as a tree

The *Person* class is the root class of the aggregation hierarchy, while *Bank*, *Engine*, *University* and *Computer* are leaf classes. The other classes; *Vehicle*, *Course* and *Company* are intermediate classes. We can create four possible paths from the root class to its leaf classes as follows.

Path 1: *Person* → *Vehicle* → *Company* → *Bank*

Path 2: *Person* → *Vehicle* → *Engine*

Path 3: *Person* → *Course* → *University*

Path 4: *Person* → *Computer*

When we specify an object of the *Person* class, using the forward traversal method can retrieve the corresponding objects of the nested classes for each path. It is also noticeable from Figure 1.1 that the join classes are the *Person* class and the *Vehicle* class. The Path 2 can be reduced to *Vehicle* → *Engine* because the corresponding objects of the *Vehicle* class from Path 1 are sufficient for further retrieval of objects from the classes of Path 2. We classify the queries by the following factors.

1. The class traversal methods from the predicate class to the target class are as follows.

$F(A,B)$: Forward traversal from class *A* to Class *B*.

$R(A,B)$: Reverse traversal from class *A* to class *B*.

2. The number of paths involved for the predicate class and the target class are.

SP: The predicate and the target class are on the same path.

MP: The predicate class and the target class are on different paths.

The examples of queries are given from the classification above. The symbol *PC*, *TC* and *JC* are denoted for the predicate class, the target class and the join class respectively.

Q1: Retrieve persons who own cars that are made by the companies that connect to Bangkok Bank.

($R(PC,TC)$, SP)

Q2: Retrieve banks connected by the companies that manufacture cars owned by persons at the age of 40.

($F(PC,TC)$, SP)

Q3: Retrieve engines of the cars owned by the persons who take course at Chulalongkorn University.

($R(PC,JC)$, $F(JC,TC)$, MP)

Most indexing techniques can tackle the problem such as Q1 when the predicate is specified on the indexed attribute of the leaf class and the target class is the root class. A few techniques are proposed to eliminate the forward traversal between classes of the single path for the query Q2. Although applying the combination of various indexes can solve the query Q3, the joining between paths is still required and overhead occurred is considerable. The detail of overhead analysis will be discussed later.

The access methods of the aggregation hierarchy as a tree have been proposed recently. Direct Access to Terminal Virtual Path [28] is as follows. For each object in the root class *Person*, there will be corresponding objects in leaf classes *Bank*, *Engine*, *University* and *Computer*. Associated objects in leaf classes are stored together as if there were a path between them. This path is called Terminal Virtual Path (TVP). Therefore, the information in TVP consists of OIDs of the leaf classes that associate with

the object in the root class. OID of the object in the root class is stored with the associated TVP as an entry in the linking file structure. Index can be created on simple attributes of the root class and map to the associated entries in the linking file. This access method shows that linking between objects in the root class and corresponding objects in the leaf classes stored in TVP can reduce cost of intermediate classes traversal. However, it is only suitable for the query that the predicate class and the target class are on leaf classes or the root class. Virtual Path Signature [29] is proposed to handle multi key indexing. For each aggregation of objects from the class schema from Figure 1.1, associated objects in leaf classes are stored together in a virtual path called Terminal Virtual Path (TVP) and associated objects in non-leaf classes are stored in a virtual path called Non-Terminal Virtual Path (NTVP). Signature is generated for objects in TVP and NTVP. The Virtual Path Signature shows significant improvement in retrieval when compared with Tree Signature [23], especially when the number of classes between the target class and the predicate class is high. However, its retrieval performance is lower when compared with the indexed attributes of the indexing techniques. Therefore a new approach should be proposed to tackle limitation mentioned above. It should have the characteristics as follows.

1. Its structure should be stored in the secondary storage other than OODB.
2. It should support traversal of classes in the aggregation hierarchy.
3. It should support associative searching.
4. It should support various kinds of queries for the aggregation hierarchy; i.e. the predicate class and the target class can be anywhere in the aggregation hierarchy.
5. Its cost model in terms of storage overhead and retrieval cost should be lower than other approaches when applied as multi paths, for example, the Path Dictionary Index [26].

In conclusion, this new approach with 5 characteristics is the motivation of this research.

1.2 The Purpose of the Research

1. To propose a new access method for aggregation hierarchy as a tree in object-oriented database.
2. To formulate cost models of the new access method in terms of storage cost, retrieval cost and update cost.

1.3 The Scope of the Research

1. An access method will be built by using the data from the object-oriented database.
2. The logical linking of classes in object-oriented database is an aggregation hierarchy as a tree.
3. The value of an attribute is a single value.
4. The access method can handle a simple predicate and a complex predicate.
5. All objects in the child class are referenced by the objects in the parent class.
6. Cost models of the new approach will be compared with that of the previous approaches.

1.4 The Benefits of the Research

1. The new access method that is suitable for the aggregation hierarchy as a tree in object-oriented database.

2. The cost model that is better than the traditional approaches.

1.5 The Method of the Research

1. Study related works.
2. Design a new access method.
3. Consider the database operation on the new structure.
4. Analyze the cost models in terms of
 - 4.1 The storage cost.
 - 4.2 The retrieval cost.
 - 4.3 The update cost.
5. Make the comparison with other models.
6. Conclusions.

The next chapter will contribute to the related theories and related works, especially various techniques of access methods.

CHAPTER 2

RELATED WORKS

This chapter contributes to the related theory and researches. The theory starts from the object data model and ends with the query processing. The related works of access methods are classified as the indexing techniques and the signature techniques.

2.1 Theory

There are many object-oriented database systems that have been developed and implemented such as Gemstone [31], O₂ [11] and etc. Object-oriented databases have been widely used because they have the capabilities to handle the complex applications, such as Computer Aided Design/Computer Aided Manufacturing (CAD/CAM) design, office automation and Very Large Scale Integration (VLSI) design. Object – oriented data model is a logical organization of the real world object (entities), constraints on them, and relationships among objects. The following is an object-oriented data model [19] and most of the concepts are accepted as the Object Database standard by the Object Database Management Group (ODMG) [2].

2.1.1 Object and Object Identifier (OID)

The uniform treatment of any real – world entity as an object simplifies the user's view of the real world. The object identifier (OID) is used to pinpoint an object to retrieve. Two methods are possible to represent OID, namely, physical address and logical address.

The method of physical address representation provides the good retrieval performance. However, as an object relocates, retrieval performance becomes worse and OID may not be unique. By using logical address representation for OIDs, called surrogate, the objects are independent on storage structures. Yet, there must be a mapping table called OID table that maps each surrogate to its physical address.

The object identifier (OID) is not reused even when the object with which it was associated is deleted from the system.

2.1.2 Attributes and Methods

Every object has a state and a behavior. The state of an object is the set of values for the attributes of the object, and the behavior of an object is the set of methods (program code) which operate on the state of the object. An attribute of an object may have a single value or a set of values. The domain of an attribute may be any class, user defined or primitive. An example of an object-oriented schema is shown in Figure 2.1 adopted from [10].

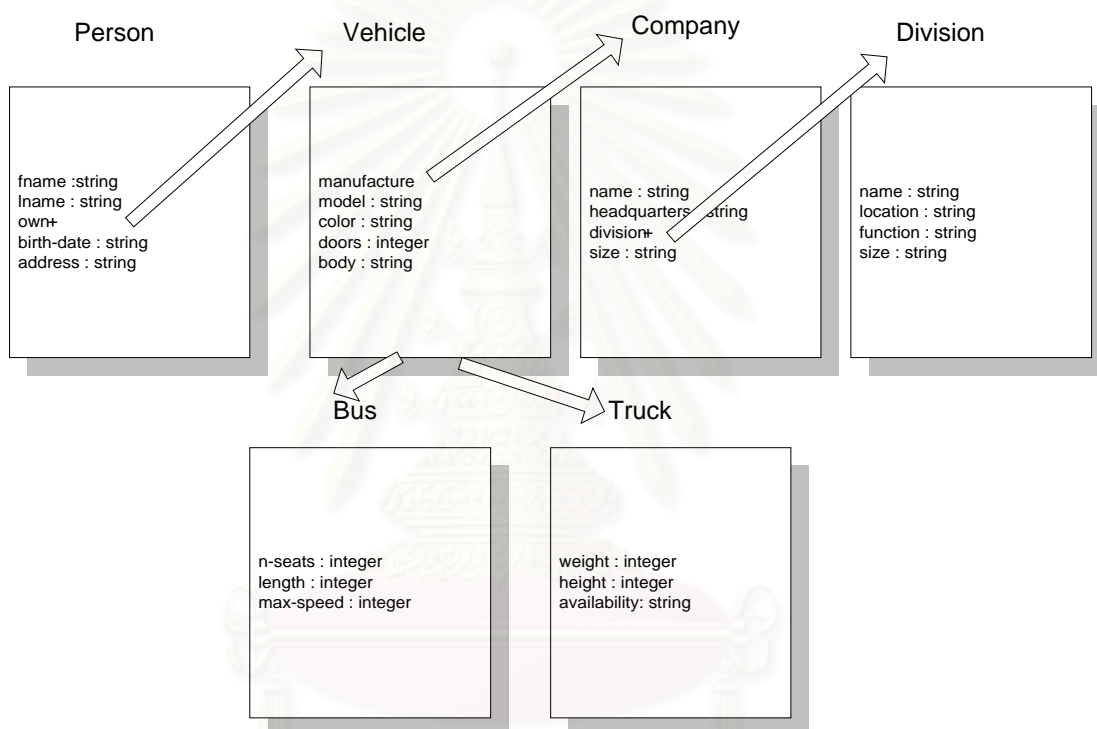


Figure 2.1 Example of an object-oriented logical schema

The domain of the *fname* attribute of the *Person* class is the primitive class string; and the value of the *fname* attribute of an instance of the *Person* class may be string “Somsak”. In contrast, the domain of the *own* attribute of the *Person* class is the *Vehicle* class; the value of the *own* attribute of a *Person* instance may then be the object identifier (OID) of several instances of the *Vehicle* class. Multi – value attributes are marked by ‘ + ‘.

2.1.3 Class

A class is specified as a mean of grouping all the objects that share the same set of attributes and methods; (there are six classes in the Figure 2.1). An object must belong to only one class as an instance of that class. The relationship between an object and its class is the instance — of relationship. A class may also be primitive. A primitive class is one which has associated instances, but which has no attribute.

The value of an attribute of an object, since it is an object, also belongs to some class. This class is called the domain of the attribute of the object.

2.1.4 Class Hierarchy and Inheritance

Object-oriented systems allow the user to derive a new class from an existing class. The new class, called a subclass of the existing class, inherits all attributes and methods from the existing class, called a superclass of the new class. The *Bus* class in Figure 2.1 is a class derived from the *Vehicle* class. It has the same attributes and methods as in the *Vehicle* class. Moreover, it has additional attributes, namely, *n- seats*, *length* and *max-speed*.

A class may have any number of subclasses. The *Vehicle* class has two subclasses: *Bus* and *Truck*. Some systems allow a class to have only one superclass, while others allow a class to have any number of superclasses. In the former, a class inherits attributes and methods from only one class; this is called single inheritance. In the latter, a class inherits attributes and methods from more than one superclass; this is called multiple inheritance. In a system that supports single inheritance, the class forms a hierarchy called a class hierarchy or inheritance hierarchy. If a system supports multiple inheritance, the class forms a rooted directed graph called a class lattice.

2.1.5 Aggregation Hierarchy

The fact that the domain of an attribute may be an arbitrary class gives rise to the nested structure of the definition of a class.

That is, a class consists of a set of attributes; the domain of some or all of the attributes may be classes with their own set of attributes, and so on. Then the definition of a class is a directed graph of classes rooted of that class. If the graph for the definition of a class is restricted to a strict hierarchy, it is called class — composition

hierarchy or aggregation hierarchy. In Figure 2.1, the aggregation hierarchy has the *Person* class as the root class and the *Division* class as a leaf class. An attribute of any class on aggregation hierarchy is logically an attribute of the root of the hierarchy, i.e. the attribute is a nested attribute of the root class. For example, in Figure 2.1, the *name* attribute of the *Company* class is a nested attribute of the *Person* class.

2.1.6 Query Processing

Banerjee [1] developed a model for queries in object-oriented databases and illustrated the model using a version of the query language implemented in ORION. A query may be formulated against an object-oriented schema, which will fetch instances of a class, which satisfy certain search criteria. A query may restrict the instances of a class to be fetched by specifying predicates against any attributes of the class. Another research on query languages [7] pointed to the characteristics of an object-oriented data model, such as object identity, complex object structure, methods, and class hierarchies, have an impact on the design of a query language. Furthermore, Kim [20] investigated cyclic query processing and developed cost model to determine the cost for each access plan generated. Given a query, there are many access plans that a database management system (DBMS) can follow to process it and produces its answer. All plans are equivalent in terms of their final output but vary in their cost, i.e. the amount of time that they need to run. A module called the query optimizer [16] can examine all alternatives and choose the plan that needs the least amount of time.

2.2 Related Works

In this subsection, several access methods of aggregation hierarchy will be briefly described. We can divide access methods into two groups, i.e. the indexing technique and the signature technique.

2.2.1 Indexing Techniques

Much research has been done and is still going on to develop well-founded data models. To be viable, not only the concept of OODB have to be supported

by an architecture that directly implements them but querying and maintaining the database should require an acceptable amount of time.

The indexing technique is considered to accelerate database operations by constructing efficient access structures on a database given a certain physical implementation of the database. Secondary index on an attribute or a combination of attributes is useful for evaluating queries on a nested class in an object-oriented database. Classic research on index [6] has been performed on aggregation hierarchy, for example, multi index, nested index, path index.

2.2.1.1 Multi Index

Multi index [6] is created for two classes that linked by the inherent pointer of the complex attribute. If there is a relation of classes from $C_1C_2C_3\dots C_n$, where C_1 is the root class and C_n is the leaf class and C_1 has an attribute whose domain is class C_2 and so on, and there are n classes in this relation, then there will be n multi index. For n^{th} multi index, index will be created on a simple attribute of class C_n and the key will link to associated OIDs of objects in class C_n . For i^{th} multi index, index will be created on a complex attribute of class C_i and the key will link to associated OIDs of objects in class C_i . If predicate is on indexed attribute of class C_n and the target is on class C_i and there is a relation from class C_i to class C_n , $(n-i+1)$ index lookup will be required. Therefore, this index is not suitable for the query when the predicate class and the target class are far away. However, multi index has the flexibility for the update because it is easy to update link between key index and associated OIDs.

2.2.1.2 Nested Index

This index [6] is created from the principle that there is a relation of classes from $C_1C_2C_3\dots C_n$, where C_1 is the root class and C_n is the leaf class. The index is created on a simple attribute of the leaf class and the key will link to associated OIDs of objects in the root class. We can see that it is suitable for the query that the predicate is specified on the indexed attribute of the leaf class and the target is on the root class. However, it is not suitable to use this index if the predicate class and the target class are

anywhere. Furthermore, the update requires the reverse traversal from the updated object to its parent objects in the root class. Therefore, the reverse pointer should be implemented in the aggregation hierarchy to support the update operation.

2.2.1.3 Path Index

Path index [6] is similar to nested index. The index is created on a simple attribute of the leaf class and this key will link to associated paths from class C_1 to class C_n . The information stored in a path is the linking of OIDs of objects from classes on the path so that if the predicate is specified on the indexed attribute of the leaf class, the target class can be any classes. Although the path index can support more queries than the nested index, it requires more storage overhead.

2.2.1.4 Index of Direct Link

Direct link [24] is the structure stored in the secondary storage. The information of an entry in this structure is OIDs of objects of the root class and OIDs of objects of the leaf class. There are two kinds of the direct link as follows.

- **Forward Direct Link**

Each entry in the direct link is OID of an object in the root class and associated OID of an object in the leaf class. The number of entries in the forward direct link is equal to the number of objects in the root class.

- **Reverse Direct Link**

Each entry in the direct link is OID of an object in the leaf class and associated OID of an object in the root class. The number of entries in the reverse direct link is equal to the number of objects in the leaf class.

Index can be created for both forward direct link and reverse direct link and the clustering technique could be considered for low retrieval cost. However, this index is only suitable when the predicate class and the target class are on the root class or the leaf class.

2.2.1.5 Path Dictionary Index

Path dictionary [21], [25], [26] is proposed in the concept of grouping all objects in the path that link to the same object in the leaf class. Information that store in an entry of path dictionary is sufficient for the traversal of objects between classes in the path. Redundancy of objects can be eliminated from this concept so that the identity index can be easily built on this path dictionary. For associative search, the attribute index will be created on the path dictionary. So, for the key index, the target object from the qualified entry can be retrieved. When compared with path index, Path dictionary index has lower cost in cost model. Furthermore, it can cover more queries in such a way that the predicate class and the target class can be any classes in the path.

2.2.1.6 Other Researches

Bertino [5] considered the usage of the path index from [6] in the framework of more general queries containing several predicates. Implicit joins over both overlapping and non-overlapping paths are analyzed to find the most suitable index along a path. A graph-theoretic approach to the path indexing [33] is proposed later. Finally, a set of parameters able to exactly model topologies of object references in object-oriented databases and their mathematical derivations are shown in [4].

Although path indexes have been proposed for efficient processing of object-oriented queries, conventional join algorithms do not effectively utilize them. Cho [20] proposed a new join algorithm called OID join algorithm that effectively utilizes path indexes in object-oriented databases.

The research from [3] extended work from [6] by providing an integrated support for queries involving both nested attributes of objects and inheritance hierarchies. It allows a query containing a predicate on a nested attribute and involving several classes in a given inheritance hierarchy to be solved with a single index lookup.

It is concluded from [3] that the nested inherited index offers the best retrieval performance (compared with multi-index and nested inherited multi index) in most cases. It is outperformed by the other organizations only when queries are mainly on the last class of the indexed path. Therefore, the nested inherited index organization should be mainly used when the number of nested predicates in queries is high.

Also the cost of modification of nested inherited index has better performance than other organizations.

Similar work, but a different technique, Uniform indexing (U-index) proposed by [13] combines the hierarchical and nested indexing scheme. It provides path indexing with better retrieval performance than the original scheme (path index) and better performance. This scheme provides in one uniform index, combined class-path-hierarchy index, and with that it is able to answer queries which are not answerable with the previous indexing schemes (nested index, path index and multi index).

Fotouhi [12] proposed a hybrid indexing technique called a generalized index, which can support class hierarchy (inheritance hierarchy) and aggregation hierarchy. This index method can share the value among the classes whose domains are identical and can be processed in parallel.

Later, the research in [10] attempted to select optimal index configuration in object oriented database from previous works [3], [6]. This research addressed the problem optimal index configuration for a single path and a path can be achieved by splitting the path into subpaths and by indexing each subpath with the optimal index organization. Algorithm for selection indexes is presented and existing indexing techniques were considered (simple index, inherited index, nested inherited index, multi index and multi inherited index). The body of the algorithm mainly consists of 3 procedures

- computes the process costs for all possible subpaths with each index organization and represent it in a matrix.
- determines the minimal costs in each row which indicates the best indexing technique for a subpath.
- determines the optimal index configuration for a path. The idea of this procedure is based on the consideration of all possible ways to recombine the original path from subpaths

Seo [32] proposed the research that is similar to [10]. Their approach is to reduce the problem to that of selecting indexes in such a way that the sum of the cost savings is maximized subject to a given storage capacity.

Several indexing schemes mentioned above have been developed to evaluate predicate to identify complex object. However, support structures called object skeletons [15] are developed for the efficient execution of traversal to retrieve the required components. Skeletons of complex objects contain only the semantic information (OIDs of the component objects and the semantic links between them) but the descriptive information of an object is stored separately from its object identifier. Traversal through the semantic links can be performed inexpensively because each node in the OID network is very small, a very large number of objects can fit in a disk page to facilitate efficient object navigation. Once a skeleton has been loaded into memory, navigation along the skeleton can be done with no further disk accesses.

Indexing techniques can efficiently support backward traversals. But this requires high storage and maintenance costs. These overheads may constrain to limit indices for multiple attributes of the classes. Therefore, on the whole, indices are maintained only for important attributes.

2.2.2 Signature Technique

The signatures can be used instead of index. Signature is rooted from applications in text databases, which require an efficient search method. The principles required by signatures are:

- The method should be fast, requiring a few disk accesses to respond to simple queries
- The method should not require rewriting on insertion
- The method should handle insertion efficiently, without the need to excessively reorganize the database

Signature is created by encoding the textual document using the hashing and superimposed coding method. By using superimposed coding (SC), the collection of document signatures gives a bit matrix, the way this bit matrix is stored affects the time on retrieval and insertion. There are several techniques about signatures, such as, sequential signature file, bit-sliced signature file and frame-slice signature file.

We can use the signature for object-oriented database as in [23] and [39] and many researchers recently interested in using it instead of using indexing scheme. The research about applying signatures for forward traversal query processing in object oriented database [39] is one of the researches that use a signature to expedite the forward traversal. Signature generation procedure through superimposed coding from data $D = \{ G.D.Hong, 25, Seoul \}$ for the signature of D , S_D , is shown in Table 2.1 adopted from [39].

Value	Hashing result
G.D.Hong	1001 0000 1000 0001
25	1000 1100 0000 0100
Seoul	1000 0000 1100 1000
Signature S_D	1001 1100 1100 1101

Table 2.1 Signature generation example (in case $b = 16$, $k = 4$)

In this process, each value of the data (this value of attribute in one class) is hashed using a function having two parameters b and k , which represents hashing size of bits and number of bits to be set as '1' respectively. The signature is then simply derived from ORing all these hashing results.

Using this signature, we can check whether the given value is in a signature or not. This checking procedure (called signature matching) selects candidate signatures to be examined. A signature S is qualified if and only if, for all bit-1 positions in the query signature, the corresponding bit positions in S are also set to 1. To evaluate

the query, we AND the query object signature with an instance's object signature and compare the result with the query object signature. If the result is the same as the query object signature, the instance may satisfy the predicate specified in the query and we need to retrieve and examine the instance. If it is not the required instance we call this matching a false drop.

In the research [39], the object signature of the referred object is stored into the referring object so nested predicates can be checked without inspecting referred objects, supporting for forward traversals. Paper [8] used the technique similar to [39], but extended object signature for the generalization hierarchy (inheritance hierarchy). This object signature consists of two parts; a reference signature and a structure signature. Like [39], a reference signature can be used to eliminate objects that do not match the nested predicates specified in the query. A structure signature can be used to eliminate objects that do not belong to the target classes. So object signature is generated from reference signature concatenated with structure signature. The research in [8] and [39] pointed out that object signatures could be used to eliminate objects that do not satisfy the predicates before we really retrieve and examine them from disk. Thus, this mechanism can greatly reduce the number of disk accesses.

Using other signature techniques also interest researchers in [22], [34] and [36]. In [36], three organizations (path index, path table, signature file with path table) based on multi attribute hashing for using in efficient evaluation of a query are considered. The researchers pointed out that a path index is not suitable for the queries that start at any intermediate positions. A path table based on hashing techniques may require the number of buckets retrieved but if signature file is used with path table, the retrieval will speed up (signature file eliminate most of the unqualified bucket). Furthermore, they showed from the experiment that storage cost for path table is the lowest (compared with path index and signature file with path table) and the retrieval cost is the lowest for signature file with path table (except for query on the last class, path index is the lowest).

For aggregation hierarchy, using the signature path dictionary [22] is extended from the path dictionary. Path dictionary is a secondary organization, which extracts the complex attribute from the database to represent the connection between objects. This structure is proposed to support efficient object traversal for nested query processing. An object signature is generated by superimposing only the bit strings generated from the simple attributes of the object (not OID referred by attribute not in the path). Instead of associating the signature files with classes, the signatures are associated with the OIDs of their corresponding objects in the path dictionary. The

comparison between signature path dictionary and signature path shows that the signature path dictionary yields a dramatic improvement on the storage, retrieval and update cost.

In [34], a new signature scheme, called the s-signature on the path dictionary is proposed to efficiently support query processing of different types of queries. The s-signature provides an efficient filtering mechanism so that accessing the database at the initial stage of query processing is unnecessary. In other word, s-signature provides an efficient access method for the path dictionary instead of the sequential scanning.

Ishikawa [17] considered retrieval of nested objects based on the set comparison operators such as \supseteq and \subseteq . The paper proposed four set access facilities for nested objects and compare their performance in terms of retrieval cost, storage cost and update cost. The combination of the signature file method and the nested index is very promising for set retrieval of nested objects.

The next chapter will describe one of the access methods of the aggregation hierarchy as a tree called Direct Access to Terminal Virtual Path (DTVP). Also, the database operation and cost models will be presented.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

CHAPTER 3

DIRECT ACCESS TO TERMINAL VIRTUAL PATH

This chapter describes the Direct Access to Terminal Virtual Path (DTVP) for query processing of the aggregation hierarchy as a tree. Its organization and database operation will be presented. Finally, the cost models in terms of the storage cost, the retrieval cost and the update cost will be formulated.

3.1 Organization

When the class schema of the object-oriented database is formed as an aggregation hierarchy as a tree, we will organize the structure to support query processing so that the access method between the root class and the leaf classes can be performed with efficiency. The necessary component used for supporting access method between the root class and the leaf classes are as follows.

3.1.1 Terminal Virtual Path

The structure of the Terminal Virtual Path (TVP) [28] comes from the concept that all leaf classes of the aggregation hierarchy as a tree is grouped together. Therefore, when objects are instantiated, objects for the associated leaf classes will be stored together.

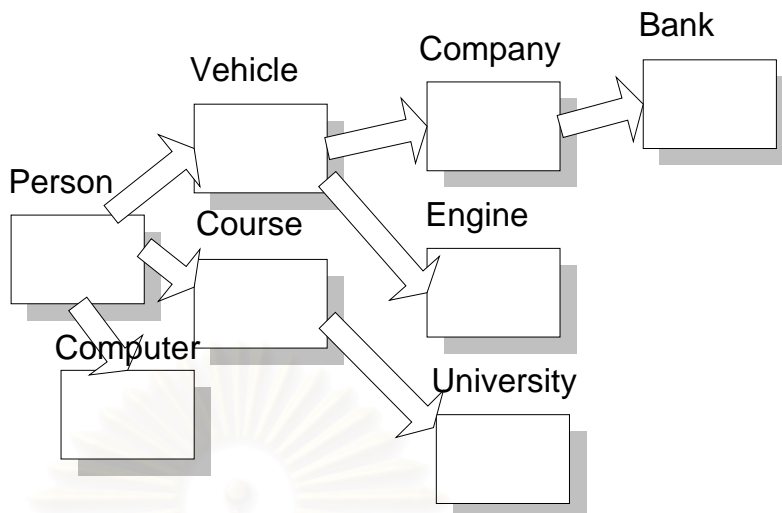


Figure 3.1 Aggregation hierarchy as a tree

From Figure 3.1, *Person* is the root class while *Computer*, *University*, *Engine* and *Bank* are leaf classes. All leaf classes will be organized as if there is a linking path between them called the Terminal Virtual Path (TVP). Therefore the class *Computer*, *University*, *Engine* and *Bank* will be grouped and their instances will be kept together in the secondary storage. If θ_i represents the object of a leaf class t_i , then the Terminal Virtual Path can be represented as follow.

Terminal Virtual Path (TVP) = $\{\theta_1, \theta_2, \theta_3, \dots, \theta_n\}$ when $\theta_1, \theta_2, \theta_3, \dots, \theta_n$ are associated OIDs of objects of the leaf classes $t_1, t_2, t_3, \dots, t_n$ respectively in the aggregation hierarchy as a tree. The structure of the Terminal Virtual Path is shown in Figure 3.2.

OID of t1	OID of t2	OID of t3	OID of tn
-----------	-----------	-----------	-------	-----------

Figure 3.2 The structure of Terminal Virtual Path

3.1.2 Root-Terminal Virtual Path Linking

To handle the direct access between objects of the root class and the associated objects in the Terminal Virtual Path, the representation will be described as follow.

$$R \rightarrow TVP = \{(\theta_1, tvp_{\theta_1})\} \text{ where}$$

$R \rightarrow TVP$ represents the linking between the object of the root class and the associated objects in Terminal Virtual Path.

θ_1 is the OID of the object in the root class and tvp_{θ_1} is the associated OIDs of the objects in the Terminal Virtual Path.

The structure of Root Class-Terminal Virtual Path linking is shown in Figure 3.3.

OID of Root Class	OID of t1	OID of t2	OID of t3	OID of tn
-------------------	-----------	-----------	-----------	-------	-----------

Figure 3.3 The structure of root class-Terminal Virtual Path linking

3.1.3 Attribute Index

The attribute index is used for fast retrieving the qualified records of the linking file structure. Index will be created on some attributes of the root class. The linking records that match the key index will be accessed so that the required target objects of the Terminal Virtual Path will be achieved.

3.2 Database Operation

The database operation in this case includes the retrieval and the update operation. The examples for these operations will be described in Section 3.2.1 and Section 3.2.2.

3.2.1 Retrieval Operation

This access method is appropriate for the query that the predicate is specified on the indexed attribute of the root class and target class is one of the leaf classes in the Terminal Virtual Path. In the research, It is assumed that the predicate is specified on the indexed attribute of the root class. Therefore, the key attribute is scanned from the attribute index to obtain the records of the linking file. For the records retrieved, the target objects are received from the target class. For example, the query "To find that Somsak own a car that manufactured by the company that connect to which bank". In this case, the predicate class is the *Person* class and the target class is the *Bank* class. If the index is created on the *Name* attribute of the *Person* class, the index will be scanned to achieve the qualified records of the linking file. Then for the qualified records, the OIDs of the target objects from the Terminal Virtual Path can be obtained to retrieve OID of the *Bank* class. Furthermore, from the structure of the Terminal Virtual Path, the associated objects can be retrieved from the other leaf classes, such as the *Engine* class, the *University* class and the *Computer* class for the object whose name is Somsak. Therefore, by using this access method, class traversal of the intermediate classes can be eliminated.

3.2.2 Update Operation

Objects of the leaf classes can be updated by specifying the object of the root class and its relevant object of the leaf class that will be modified. For example, if an object O_n of a leaf class t_n relate to object O_r of the root class the object O_n is

changed to object O'_n , the object O_1 from the linking file will be searched and update the qualified record of object O_n of class t_n to O'_n .

3.3 Cost Model

In this section, the cost model will be formulated in terms of storage cost, retrieval cost and update cost. The list of parameters is given below for the analysis.

3.3.1 The Parameters of Cost Model

Given an aggregation hierarchy as a tree, the parameters of the cost model are listed below.

- C_1 : The root class of the aggregation hierarchy T .
- C_{ni} : The i^{th} leaf class of the aggregation hierarchy T .
- nLC : The number of leaf classes of the aggregation hierarchy T .
- N_i : The number of objects in the class C_i .
- A : The attribute of the class C_1 that is selected for indexing.
- U : The number of distinct values of A .
- q : The ratio of shared attribute values between objects in the class C_1 and the value for the attribute A . ($q = N_1/U$).
- $UIDL$: The length of Object Identifier.
- P : Page size.
- pp : The size of page pointer.
- f : Average fan out of a non-leaf node.
- kl : Average length of a key value in attribute index.
- kv : Average of a key value in A .
- ll : The length of the link number.
- cl : The length of the link counter.
- h : The number of levels of non-leaf node, i.e. the height of the attribute index - 1.

Since page is a basic unit to access data in the secondary storage, so it is used for cost estimation. All lengths and sizes above are in byte.

To simplify the cost model, it is assumed that

1. All attributes have a single value.
2. All key values have the same length.
3. Retrieval and update operation are performed on the objects of the classes on the linking structure.

3.3.2 Storage Cost

To simplify the analysis, the clustering index will be created for the attribute of the root class. There are 3 main parts for the storage cost, i.e. the linking file, the non-leaf node and the leaf node of the index.

Linking file

A linking record consists of OID of the root class and nLC OIDs of leaf classes. Therefore, the size of a linking record is

$$(nLC + 1) * UIDL.$$

Each object in the root class can link directly to its Terminal Virtual Path. Therefore, the size of the linking file (SF) is

$$SF = N_1 * (nLC + 1) * UIDL.$$

The number of pages needed is

$$SFP = \lceil N_1 * (nLC + 1) * UIDL / P \rceil.$$

Non-leaf nodes

The number of non-leaf pages can be derived from

$$NLP = \lceil LO / f \rceil + \lceil \lceil LO / f \rceil / f \rceil + \dots + 1.$$

where $LO = \min(U, LP)$ and LP is the number of leaf pages for index implementation.

Leaf nodes

Due to the clustering index, the length of a leaf node record is

$$XP = kl + kv + pp + ll + cl.$$

Therefore, the number of leaf pages needed is

$$LP = \lceil U / \lfloor P / XP \rfloor \rceil.$$

The total storage cost for implementation is

$$SC = SFP + NLP + LP.$$

3.3.3 Retrieval Cost

It is assumed that the predicate is specified on the indexed attribute of the root class and the target class is any classes of the leaf classes. The query is tackled by searching through non-leaf nodes and leaf nodes of the attribute index. Then, the pointer from the leaf node will point to the target records of the linking file. The number of pages accessed is

$$RC = h + 1 + \lceil q * (nLC + 1) * UIDL / P \rceil.$$

When the leaf page is less than the page size.

3.3.4 Update Cost

The update operation can be categorized as follows.

- A. Update linking between objects of the root class and objects of the leaf class
- B. Update the attribute index for objects of the root class

Case A: Update linking between objects of the root class and objects of the leaf class.

In this case, an object O_r of the root class that previously points to an object O_n of a leaf class C_n is changed to point to an object O'_n of the leaf class C_n . It is apparently that the information of the involved linking record must be modified. The attribute index of the object in the root class can be used to search for the qualified linking record of the linking file and then modify that record. Therefore, the update cost consists of cost of the index scanning, cost of linking record retrieval and cost of linking record modification.

$$UC = h + 1 + \lceil q * (nLC + 1) * UIDL / P \rceil + \lceil (nLC + 1) * UIDL / P \rceil.$$

When the leaf page is less than the page size.

Case B: Update the attribute index for objects of the root class.

In this case, the involved objects of the root class will be affected with the modified indexed attribute. Since two index scans are needed, the number of page accesses for the update is:

$$UC = 2 * (h + 2 * \lceil XP / P \rceil).$$

It is noticeable that the Direct Access to Terminal Virtual Path has the limit for the query that the target class is not the leaf class. In the next chapter, more flexible access method called Virtual Path Signature (VPS) will be presented.

CHAPTER 4

VIRTUAL PATH SIGNATURE

This chapter proposes the application of the signature technique using with the virtual path called the Virtual Path Signature. The database operations and the cost models in terms of the storage cost, the retrieval cost and the update cost will be formulated thoroughly.

4.1 Organization

This access method can support various kinds of queries by using the signature technique on the virtual path [29]. The component of the Virtual Path Signature is as follows.

4.1.1 Terminal Virtual Path

The structure of the Terminal Virtual Path is the same as mentioned in Section 3.1.1. Therefore, the Terminal Virtual Path (TVP) = $\{\theta_1, \theta_2, \theta_3, \dots, \theta_m\}$ when $\theta_1, \theta_2, \theta_3, \dots, \theta_m$ are associated OIDs of objects of the leaf classes $t_1, t_2, t_3, \dots, t_n$ respectively in the aggregation hierarchy as a tree. The structure of the Terminal Virtual Path is shown in Figure 4.1.

OID of t1	OID of t2	OID of t3	OID of tn
-----------	-----------	-----------	-------	-----------

Figure 4.1 The structure of the Terminal Virtual Path

4.1.2 Non-Terminal Virtual Path

If θ_{nti} represents the object of a leaf class nti , then the Non-Terminal Virtual Path can be represented as follow.

The Non-Terminal Virtual Path (NTVP) = $\{\theta_{nt1}, \theta_{nt2}, \theta_{nt3}, \dots, \theta_{ntm}\}$ when $\theta_{nt1}, \theta_{nt2}, \theta_{nt3}, \dots, \theta_{ntm}$ are associated OIDs of objects of the non-leaf classes $nt1, nt2, nt3, \dots, ntm$ respectively in the aggregation hierarchy as a tree. The structure of the Non-Terminal Virtual Path is shown in Figure 4.2.

OID of nt1	OID of nt2	OID of nt3	OID of ntm
------------	------------	------------	-------	------------

Figure 4.2 The structure of the Non-Terminal Virtual Path

4.1.3 Virtual Path Signature

Non-Terminal Virtual Path and its associated Terminal Virtual Path are kept in a structure called a Virtual Path. It is similar to how to store tree of object instantiation so that the object traversal can be eliminated. However, the associative searching is also a crucial factor for query processing. It is possible that any attributes of any classes in the aggregation hierarchy may be queried but impossible to create index for every attribute. Therefore, the signature file is an alternative approach to manage multi key indexing. Signatures will be generated for all objects in the Terminal Virtual Path and all objects in the Non-Terminal Virtual Path. Then, the Virtual Path structure and its signature will be stored in a structure called the Virtual Path Signature.

The definition of the Virtual Path Signature is as follows. For the aggregation hierarchy of classes that form a tree, there will be only one signature file. The number of entries in this signature file is equal to the number of objects in the root class. Each entry consists of $\langle \text{Sig(TVP)}, \text{TVP structure}, \text{Sig(NTVP)}, \text{NTVP structure} \rangle$ where Sig(TVP) is the signature of the Terminal Virtual Path and Sig(NTVP) is the

signature of the Non-Terminal Virtual Path. The structure of the Virtual Path Signature is shown in Figure 4.3.

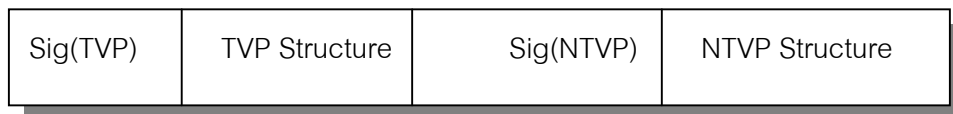


Figure 4.3 The structure of the Virtual Path Signature

The signature of the Terminal Virtual Path is generated as follows.

1. The signature of the Terminal Virtual Path is obtained by superimposing the object signatures for objects on the Terminal Virtual Path of associated aggregation hierarchy.
2. The signature of an object is generated by superimposing the signatures of all of its simple attributes.
3. The signature of a simple attribute is obtained by hashing on the attribute values.

The generation of the signature of the Terminal Virtual Path is shown in Figure 4.4.



Figure 4.4 The signature of the Terminal Virtual Path

The process how to generate the signature of the Non-Terminal Virtual Path is similar to that of above but it performs with objects on the Non-Terminal Virtual Path.

4.2 Database Operation

The operation performed on the database will be presented such as the retrieval operation and the update operation.

4.2.1 Retrieval Operation

A query of which values to be searched is transformed into a query signature S_Q . If the query Q is on classes of TVP, S_Q will be compared with every signature stored in $\text{Sig}(\text{TVP})$; otherwise S_Q will be compared with every signature in $\text{Sig}(\text{NTVP})$. When the signature matches with the query signature, It will verified if that entry of the signature file is not false drop by retrieving information in OODB using OID of the predicate class stored in the Virtual Path. If it is a qualified object, the information will be retrieved from the database by using the OID obtained.

From the characteristic of the Virtual Path Signature, searching in OODB for the aggregation hierarchy as a tree can be performed with flexibility when compared with indexing techniques as follows.

1. Associative searching can be performed with multi key on any attributes due to the signature.
2. The cost of object traversal can be reduced due to the Virtual Path structure.
3. The storage overhead of the signature is lower when compared with the index structure.

4.2.2 Update Operation

The update operation can be categorized as follows.

- A. Update the simple attribute of an object.
- B. Update the complex attribute of an object.

Case A: Update the simple attribute of an object.

In this case, a simple attribute will be changed to a new value. Since the simple attribute is used to generate the signature of an object, regenerating of the object signature is required. If the modified object is in the class of the Terminal Virtual Path, the signature of Terminal Virtual Path has to be regenerated. Similarly, the signature of Non-Terminal Virtual Path has to be regenerated if the modified object is in the class of Non-Terminal Virtual Path

Case B: Update the complex attribute of an object.

In this case, a complex attribute will be changed so that the associated entries of the signature file will be modified.

4.3 Cost Model

In this section, the cost model in terms of storage cost, retrieval cost and update cost for the Virtual Path Signature will be formulated. The parameters will be given below for the analysis.

4.3.1 The Parameters of Cost Model

Given an aggregation hierarchy as a tree, the parameters of the cost model are listed below.

nNLC : The number of non-leaf classes of the aggregation hierarchy.

nLC : The number of leaf classes of the aggregation hierarchy.

N_i : The number of objects in the class C_i .

S : The size of a signature.

$UIDL$: The length of Object Identifier.

SZ : The average size of an object.

E : The average size of an entry in the signature file.

K_S : The average size of a signature file.

R : The average matching rate of a query signature.

P : Page size.

H_i : The number of ancestor classes from the root class to i^{th} class.

N_{par} : The average number of parent objects for an object.

4.3.2 Storage Cost

The structure of the Virtual Path Signature consists of the signature of Terminal Virtual Path, the TVP structure, the Signature of Non-Terminal Virtual Path and the NTVP structure. Therefore, the size of an entry of the signature file is:

$$E = S + (nLC * UIDL) + S + (nNLC * UIDL).$$

$$E = 2S + UIDL * (nLC + nNLC).$$

Thus, the size of the signature file is:

$$K_S = N_1 * (2S + UIDL * (nLC + nNLC)).$$

The storage cost of the Virtual Path Signature is:

$$SC = \lceil K_S / P \rceil.$$

4.3.3 Retrieval Cost

The cost of retrieval cost consists of the following.

- Cost of scanning the signature file to check the query signature.

- Cost of accessing the candidate objects for the matching of the signature.

It is assumed that the candidate objects are in different pages.

Therefore, the number of page access is:

$$RC = \lceil K_S / P \rceil + (R * N_p) \lceil SZ / P \rceil.$$

When N_p is the number of objects in the predicate class.

4.3.4 Update Cost

The update cost is formulated according to the category in Section 4.2.2.

I assume that the modified entries of the signature file are in different pages.

A. Update the simple attribute of an object.

It is assumed that the modified object is in i^{th} class of the aggregation hierarchy. The update cost consists of the following.

1. Cost of scanning the signature file to find the specified object.
2. Cost of accessing the modified object and associated objects to re-compute the object signature and the signature for the virtual path.
3. Cost of writing the associated entries back to the signature file.

$$UC = \lceil K_S / P \rceil + N_{par}^{Hi} * nC * \lceil SZ / P \rceil + N_{par}^{Hi}.$$

when $nC = nNLC$ if the modified object is in the class of NTVP,

$nC = nLC$ if the modified object is in the class of TVP.

B. Update the complex attribute of an object.

The update cost consists of the following

1. Cost of scanning the signature file to find the specified object.
2. Cost of accessing the modified object and all associated objects to modify the entries in the signature file.
3. Cost of writing the associated entries back to the signature file.

The modification of an entry in the signature file consists of modification of TVP or NTVP structure and computation of the new signature for the virtual path.

$$UC = \lceil K_s / P \rceil + N_{par}^{Hi} * (nLC + nNLC) * \lceil SZ / P \rceil + N_{par}^{Hi} .$$

Although, the Virtual Path Signature is flexible for the query that the predicate can be specified on any attributes of a class, Its retrieval cost is still high when compared with the indexing technique. The next chapter will contribute to the new indexing technique called the branch index. The condition of reference sharing among objects will be considered in the formulation of cost models.

CHAPTER 5

BRANCH INDEX

This chapter contributes to the new access method using the indexing technique called the Branch Index. Its organization and database operations are thoroughly explained. Like the previous chapters, the cost models in terms of the storage cost, the retrieval cost and the update cost are formulated.

5.1 Organization

In this access method [30], various kinds of queries can be solved, for example; the predicate class and the target class can be any classes of the aggregation hierarchy as a tree. Furthermore, the reference sharing between objects is considered, Several terms used in the analysis are defined as follows.

5.1.1 Definitions

In this section, several definitions will be given for the Branch Index.

Definition 1:

For an aggregation hierarchy as a tree, if C_j is a non-leaf class or a leaf class and C_n is a leaf class and it is accessible from the class C_j , a relation from the class C_j to the class C_n will be called a *branch* in the aggregation hierarchy.

Example 1: Let us consider the aggregation hierarchy as a tree in Figure 5.1. The following are possible branches for the aggregation hierarchy.

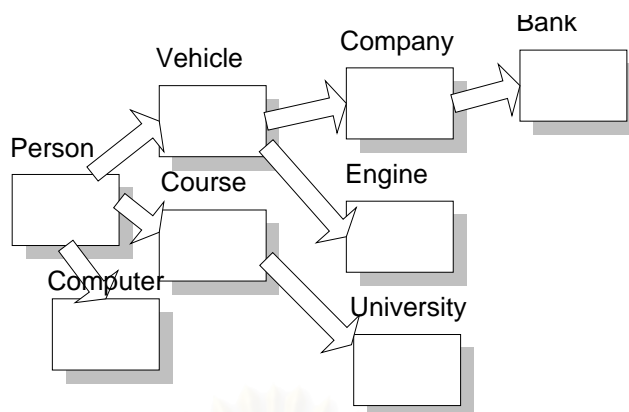


Figure 5.1 Aggregation hierarchy as a tree

$B_1: Person \rightarrow Vehicle \rightarrow Company \rightarrow Bank$

$B_2: Course \rightarrow University$

$B_3: Engine$

$B_4: Computer$

Note that each class in a branch cannot be a member of other branches.

The other possible branches can be as follows.

$B'_1: Company \rightarrow Bank$

$B'_2: Vehicle \rightarrow Engine$

$B'_3: Course \rightarrow University$

$B'_4: Person \rightarrow Computer$

Definition 2:

The *branch length* indicates the number of classes in a branch. A branch will be called a *complete branch* if its branch length is greater than one. If there is only one class in a branch, it will be called an *incomplete branch*.

Example 2: From Example 1, B_1 and B_2 are complete branches because their branch length is 4 and 2 respectively. B_3 and B_4 have only one class, so they are incomplete branches.

Definition 3:

For an aggregation hierarchy as a tree, the longest branch in the aggregation hierarchy is called the *main branch*. If L is a set of leaf classes in the aggregation hierarchy, the main branch will start from the root class C_1 to a leaf class C_n ; when C_n is a member in L .

Example 3: The main branch from Example 1 is B_1 because it is the longest branch and it starts from the root class *Person* to the leaf class *Bank*.

Definition 4:

For an aggregation hierarchy as a tree, if C_k is a class in the main branch B_i that references a class C_m , which is not on B_i , there will be a *child branch* of B_i starting from class C_m to its accessible leaf class in L . The class C_k will be called a *join class*. Therefore, a join class is a class of a branch that can link to its child branches.

Example 4: Let us consider the aggregation hierarchy as a tree in Figure 5.1 and the example branches in Example 1. The branch B_1 can link to the branch B_3 by the join class *Vehicle*. The branch B_2 and B_4 are linked to B_1 by the join class *Person*. Therefore, B_2 , B_3 and B_4 are child branches of B_1 .

Definition 5:

For a branch B_i of an aggregation hierarchy as a tree, if its child branch is an incomplete branch, this child branch will be called a *leaf branch* of B_i .

Example 5: From Example 2, the branch B_3 and B_4 are incomplete branches. Since B_3 and B_4 are child branches of the branch B_1 , they will be leaf branches of B_1 .

5.1.2 Algorithm of Branch Generation

The purpose of the algorithm is to generate the minimum number of complete branches. The smaller number of complete branches, the smaller joining between them.

Given an aggregation hierarchy as a tree and L is a set of leaf classes in the aggregation hierarchy. The procedure of the algorithm is as follows.

1. Find the main branch by considering a leaf class C_{ni} in L that has the maximum number of classes between the root class C_r and the leaf class C_{ni} .
2. If C_{ni} is the result of the leaf class in the main branch from 1, then the new set $L = L - \{C_{ni}\}$.
3. Repeat step 4 and 5 while L is not empty.
4. Consider a join class in an existing branch B_i and find the new longest branch from the child class of that join class of B_i . If a branch B_j is a child branch of B_i and C_{nj} is the ending class of the branch B_j
 - 4.1 If the branch length of B_j is greater than 1, then the new set $L = L - \{C_{nj}\}$.
 - 4.2 If the branch length of B_j is equal to 1, then the new branch is a leaf branch and will be combined to B_i . The new set $L = L - \{C_{nj}\}$.
5. Go to step 3.

An example of this algorithm is as follows:

In Figure 5.1, $L = \{Bank, Engine, University, Computer\}$. It is apparently that the longest branch is from the *Person* class to the *Bank* class. Therefore, the main branch will be generated and *Bank* will be deleted from L . The new set of $L = \{Engine, University, Computer\}$.

Since *Course* is the child of the *Person* class in the main branch, the new branch will start from the *Course* class to the candidate leaf classes in L . So the new branch is generated from the *Course* class to the *University* class. For the remaining classes in L , the *Engine* class and the *Computer* class are direct child classes of the join class *Vehicle* and *Person* respectively. Since they are incomplete branches, *Engine* and *Computer* will be parts of the main branch. When set L is empty, the branch generation will be terminated.

5.1.3 Branch Index Organization

The architecture of the Branch Index is shown in Figure 5.2. The Branch Index is a separate structure from the object-oriented database and it is stored in the secondary storage. After using the algorithm of branch generation, the number of branches and the corresponding classes will be obtained. A set of attribute indexes and

identity indexes is on top of the branch. The Branch Index consists of the following components.

Branch information

The information in the branch is OIDs linkage of objects for the classes in the branch. Therefore, the class traversal can be handled in the branch information instead of traversal in the database. In case of any child branches, the OIDs and pointers of the parent branch are also included as the information of the child branch. So, the traversal from the child branch to its parent branch can be easily managed.

Attribute Index

While the branch information can facilitate traversal among objects of classes in the branch, it does not support predicate evaluation that involves searching the object meeting the conditions specified on their attribute values. To facilitate the associative searching, attribute indexes should be used to map attribute values to OIDs in the branch information. For example, to tackle the query “Find the person who own the vehicle manufactured by the company that connect to Bangkok Bank”, the attribute index should be created for the *Name* attribute of the *Bank* class. To find the target object of the *Person* class, the attribute index of the *Name* attribute of the *Bank* class is scanned to obtain the qualified OIDs of the *Bank* class and the entry location of the branch information that store those OIDs. Then, the OIDs linkage in the branch information is used to retrieve the qualified OIDs from the target class.

Identity Index

Instead of creating the index by mapping the value of simple attributes to OIDs in the branch information, the identity index uses the values of complex attributes. Therefore, the branch information can be obtained with a given OIDs by using the identity index. Since identity search is important for retrieval and update, the identity index can reduce the cost for retrieval and update operations.

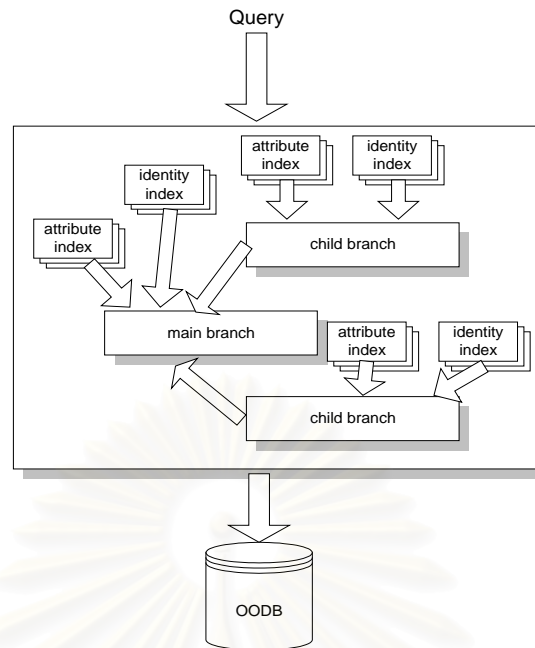


Figure 5.2 The Branch index

5.1.4 Details of the Branch Information

For a complete branch B_i of the aggregation hierarchy as a tree, there is a relation from the starting class to the ending class of this branch. When objects are instantiated, in logical view, the objects of the starting class point to their child objects until the objects of the ending class. The linking of objects is represented with linking of their OIDs or OIDs linkage. Therefore, it is much faster to traverse by using OIDs linkage in a branch than objects in the database. The necessary information that should be kept in a complete branch consists of the following:

- OIDs of objects of the classes in the branch and the pointers to their child objects.
- OIDs of the parent objects for the branch, in case it is not the main branch, and the corresponding pointers to the parent branch.
- OIDs of the leaf branch.

When several objects in one class reference the same object of the child class, it is called the reference sharing. Therefore, OIDs linkage should be kept to save the storage in case of the reference sharing. Figure 5.3 shows an example of object instantiation and the reference sharing by using the information from Figure 5.1.

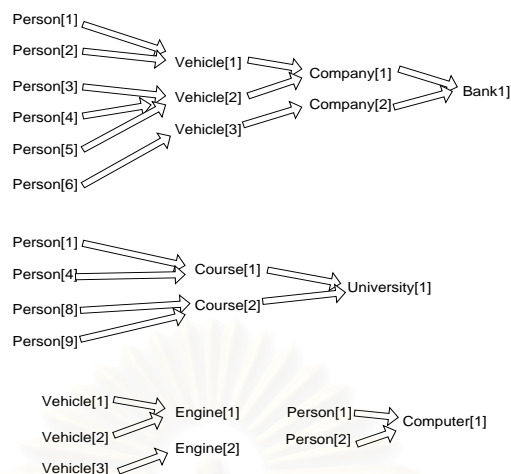


Figure 5.3 An example of object instantiation

From Figure 5.3, the i^{th} object of the *Person* class will be denoted as *Person*[i]. OIDs of objects of other classes will use the same notation. It is noticeable that *Person*[1] and *Person*[2] reference the same object *Vehicle*[1].

From the algorithm of branch generation presented earlier, two complete branches are obtained as the result as follows.

- The main branch that starts from the *Person* class to *Vehicle*, *Company* and *Bank*. Since the *Computer* class and the *Engine* class are leaf branches of the main branch, they are also part of the main branch.
- The child branch that starts from the *Course* class to the *University* class.

To cope with the reference sharing, the concept that is similar to that of the path dictionary is used for storing information. All ancestor objects of an object in the ending class of the branch will be kept as an entry of the branch information. For example, for the object *Bank*[1] of the ending class, the entry of the main branch consists of *Person*[1] to *Person*[6], *Vehicle*[1] to *Vehicle*[3], *Company*[1] to *Company*[2] and *Bank*[1]. All linkages between objects are also kept, for example, pointer between *Person*[1] and *Vehicle*[1], pointer between *Person*[2] and *Vehicle*[1] and so on. Since the leaf branch *Engine* and *Computer* are parts of the main branch, their objects will correspond to the main branch. Therefore, *Engine*[1] will be linked from *Vehicle*[1] and

Vehicle[2]; Engine[2] will be linked from *Vehicle[3]*. Finally, *Computer[1]* will be linked from *Person[1]* and *Person[2]*.

5.2 Implementation

The structure of an entry of a branch is shown in Figure 5.4.

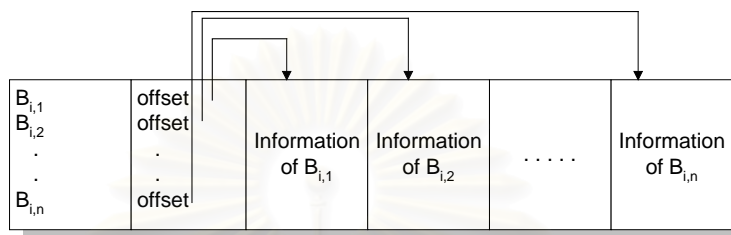


Figure 5.4 The structure of an entry of a branch

From Figure 5.4, $B_{i,1}$ denotes the starting class of i^{th} branch while $B_{i,n}$ denotes the ending class of i^{th} branch. It is assumed that there are n classes that have relation in i^{th} branch. The relation is in the form that $B_{i,1}$ references $B_{i,2}$ and $B_{i,2}$ references $B_{i,3}$, ..., $B_{i,n-1}$ references $B_{i,n}$. The offset for each class points to the location of 1st OID of the object in that class, for example, the offset of $B_{i,1}$ locates the address of 1st OID of object of the starting class. The example of an entry of the main branch is shown in Figure 5.5.

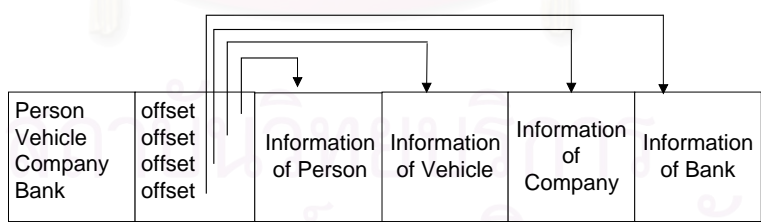


Figure 5.5 An example of the structure of an entry of the main branch

There are four classes of the main branch and their associated information for each class. The offset will point to the first entry of the information for that class. Therefore, given a specified class, the information can be determined comfortably. The detail implementation of information for each class of a branch is shown in Figure 5.6.

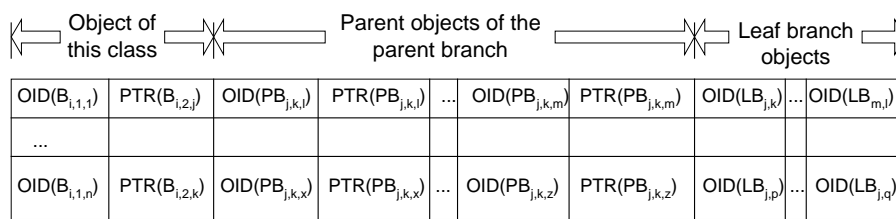


Figure 5.6 The structure of an information of a class in an entry of a branch

From Figure 5.6, $B_{i,1,1}$ and $B_{i,1,n}$ denotes the 1st object and the nth object of the starting class of the ith branch respectively. The n objects that belong to the starting class of the ith branch may point directly or indirectly to the same object of the ending class of the ith branch. Each object of the starting class is implemented as a record that consists of members as follows.

1. OID of the object itself.
2. Pointer to OID's child object.
3. Multiple pairs of OID's parent object and its pointer to the parent branch (except the main branch).
4. Multiple OIDs of leaf branch objects for the starting class in case that the starting class has leaf branches.

In general, for the second class to the class before the ending class of a branch, a record for each object of the class consists of members as follows.

1. OID of the object itself.
2. Pointer to OID's child object.
3. Multiple OIDs of leaf branch objects for the class in case it has leaf branches.

Finally, for the ending class, the information will be only OID of the ending class. The number of OIDs for the ending class will be only one for each entry of the branch. Figure 5.7 shows an example of the information of the class *Person*, *Vehicle*, *Company* and *Bank* for an entry of the main branch.

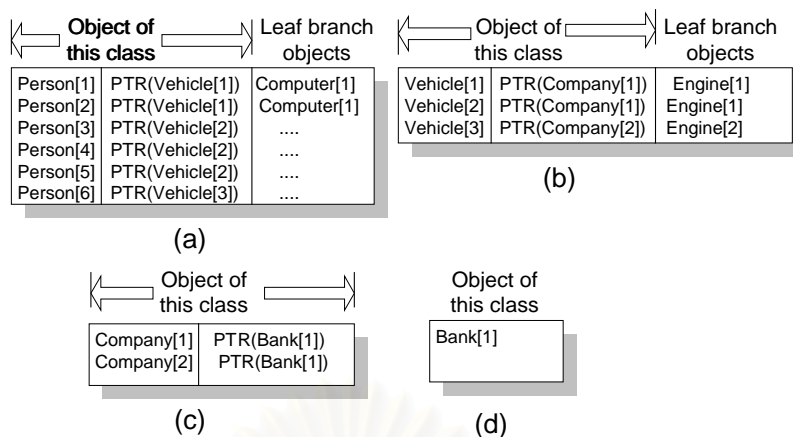


Figure 5.7 An example of the information for all classes of an entry of the main branch

From Figure 5.7, there is no information of the parent objects and the parent pointers because it is the ancestor branch of all branches. However, for the other branches, OID's parent objects and their pointers have to be kept as mentioned earlier. An example of the information for the child branch is shown in Figure 5.8.

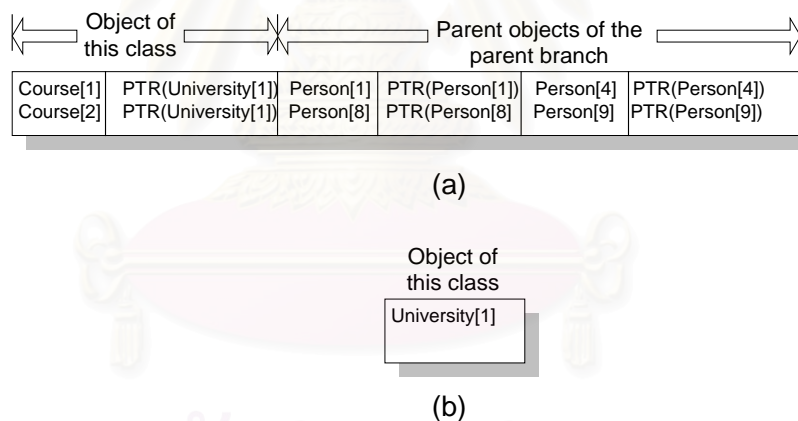


Figure 5.8 An example of the information for all classes of an entry of the child branch

The information of the *Course* class and the *University* class is shown in Figure 5.8(a) and Figure 5.8(b) respectively. Notice that this child branch has no leaf branch for all classes of the branch. However, since it is the child branch of the main branch, the OIDs of the parent objects and pointers to the main branch have to be stored.

At present time, the price of the media storage is decreasing and the capacity of the storage is increasing. Therefore, the storage overhead of an access

method is not significant when compared with the retrieval performance. The branch information will be created for every branch generated in case that the predicate class and the target class can be any classes in the aggregation hierarchy. It will be stored sequentially on the secondary storage. However, if it is known exactly where the predicate class and the target class are, the branch information can be created for the branches involved. An entry of the branch information is not allowed to cross page boundaries unless its size is greater than the page size. Free space directory is required for each page to inform the free space left. If there is not space enough left for an entry of the branch, the new page will be allocated. Therefore, the free space directory will be stored before the branch information.

The data structure that is used to model the various indexes is based on tree-structures, such as B^+ -trees. The format of a non-leaf node for the identity index is similar to that of the attribute index. Figure 5.9(a) and Figure 5.9(b) shows the format of a non-leaf node for the identity index and the attribute index respectively.

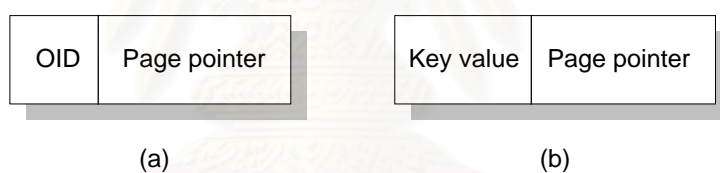


Figure 5.9 Non-leaf node record of the identity index and the attribute index

The format of a non-leaf node record of the identity index consists of OID and page pointer. The page pointer contains the address of the next level non-leaf page of the OID or the address of the leaf page of the OID. The format of a non-leaf node record of the attribute index is similar to that of the identity index. Key value is used for the attribute index instead of OID used for the identity index.

The format of a leaf node record of the identity index and the attribute index is shown in Figure 5.10.

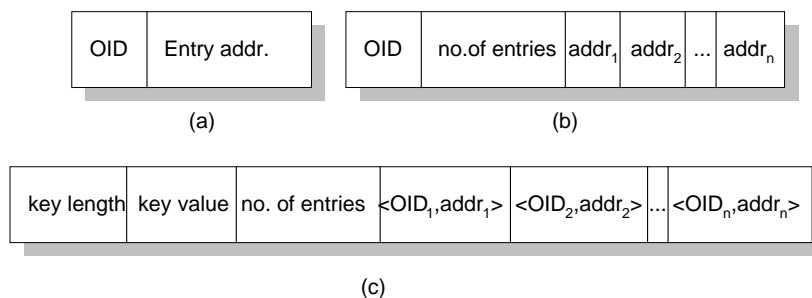


Figure 5.10 Leaf node record of the identity index and the attribute index

The identity index is created for all objects for each class of a branch. Therefore, if there are n classes involved in a branch, there will be n identity indexes for each class in the branch. A leaf node record for the identity index of any classes is shown in Figure 5.10(a). However, for the class that is a leaf branch, the identity index is shown in Figure 5.10(b). The leaf node for the attribute index is shown in Figure 5.10(c) of the corresponding OIDs and addresses for the indexed attribute.

5.3 Retrieval and Update Operation

In this section, the retrieval operation and the update operation are discussed on the Branch Index.

5.3.1 Retrieval Operations

The aggregation hierarchy as a tree shown in Figure 5.1 is used to discuss the retrieval operation. As mentioned in Section 5.1, there will be two branches generated when using the algorithm of branch generation. Therefore, B_1 and B_2 will represent the main branch and the child branch respectively. A query that involves the predicate class and the target class can be classified as follows.

- The predicate class and the target class are on the same branch.
- The predicate class and the target class are on different branches.

A. The predicate class and the target class are on the same branch

In this case, the predicate class and the target class can be any classes on the branch. For example, to find the owner of the car manufactured by the company that connected to Bangkok bank. Therefore, the predicate class is the *Bank* class and

the target class is the *Person* class of branch B_1 . In this case, if an index is created on the *Name* attribute of the *Bank* class, the qualified entries of the branch that associate with the indexed key can be determined. Since the information of an entry of a branch consists of OIDs of every class in the branch, OIDs of the objects in the *Person* class can be easily retrieved. Furthermore, if the target objects are on any classes of the branch, the OIDs of the objects for those classes can also be easily retrieved. Speaking about the leaf branch, it is also a part of a complete branch. The query that the predicate class or the target class are on the leaf branch is similar to that of discussion above, for example, to find the manufacturer of the car that own by the person who use the computer with OS UNIX. Therefore, the leaf branch, in this case, is the *Computer* class that is a part of the main branch B_1 . The predicate class is the *Computer* class and the target class is the *Company* class of branch B_1 . If an index is created on the *Name* attribute of the *Computer* class, this index can be used to find the qualified entries of the branch B_1 and access the qualified OIDs from the *Company* class. It can be concluded that if the predicate class and the target class are on the same branch and the predicate is specified on the indexed attribute, the qualified OIDs of the target class can be accessed by scanning the indexed attribute. Several attribute indexes can be created with low storage overhead because the overhead occurs only for the non-leaf node records and leaf node records of the attribute indexes.

B. The predicate class and the target class are on different branches

In this case, the predicate class and the target class occurs on different branches, for example, a predicate is on a class of branch B_1 and the target is on a class of branch B_2 . From Figure 5.1, the query "to find the university of the person who own the car manufactured by the company that connect to Bangkok bank" is an example above. Therefore, the predicate occurs on the *Bank* class of the main branch B_1 and the target class is the *University* class of the child branch B_2 . The main branch B_1 connects to its child branch B_2 by the *Person* class. If an index is created on the *Name* attribute of the *Bank* class, the OIDs of the *Person* class from the qualified entries of the main branch will be obtained. Then, the forward traversal technique can be used from each qualified OIDs of the *Person* class to the *Course* class and the *University* class of

branch B_2 and access the qualified objects from the *University* class. Also, an alternative approach is to scan the identity index of the *Person* class for the qualified objects on the branch B_2 to access the target objects of the *University* class from the qualified entries of branch B_2 . On the contrary, the query "to find the bank that is connected by the manufacturer of the car own by the person who take course at Chulalongkorn University" is somewhat different. Although the predicate class and the target class are on different branches, in this case, the join class cannot be used for the reverse traversal. If an index is created on the *Name* attribute of the *University* class of branch B_2 , this index can be scanned to obtain the qualified entries of the branch B_2 . Because the branch B_2 is the child branch of B_1 and the information of branch B_2 consists of OIDs and the addresses of the parent branch B_1 , these information can be used to determine the qualified entries of branch B_1 and retrieve the OIDs from the *Bank* class. Therefore, it can be concluded that the traversal from the child branch to its parent branch can be achieved easily by using the information stored in the child branch. However, information is not stored from the parent branch to its child branch because the forward traversal method can be used from the join objects to the target objects directly.

5.3.2 Update Operations

Figure 5.1 and Figure 5.3 are used to discuss the update operations. The update operation is considered only updating the complex attribute because it reflects the information stored in the branch. The update operations can be specified as follows.

- Update the reference between the parent object and the child object on the same branch.
- Update the reference between the parent object and the child object on different branches.

A. Update the reference on the same branch.

In this case, the parent object and its child object are on the classes of the same branch. It is assumed that an object O of class C changes the reference from an object O' of class C' to an object O'' of class C' . The identity index of class C' has to be searched to find the entries that associate with object O' and O'' . Furthermore, It is

assumed that E_1 and E_2 are the corresponding entries for O' and O'' respectively. If E_1 is equal to E_2 , the pointer that O points will be changed from O' to O'' . However, when E_1 is not equal to E_2 , OIDs and the pointers of class C and its ancestor classes that associate with object O' in E_1 have to be deleted and add these information in the entry E_2 that associates with object O'' . Also, the information stored in the child branch for the moved class has to be updated. Meanwhile, the associated identity indexes have to be updated. For example, if $Vehicle[1]$ that references $Company[1]$ changes to $company[5]$, the entry of a branch for $Company[1]$ and $Company[5]$ has to be searched. If the entries are different, $Vehicle[1]$ and the pointer to $Vehicle[1]$ will be deleted from the entry of $Company[1]$ and add this information in the entry of $Company[5]$. Furthermore, $Person[1]$, $Person[2]$ and corresponding pointers have to be moved from the entry of $Company[1]$ to the entry of $Company[5]$. All associated leaf branch objects for $Vehicle[1]$, $Person[1]$ and $Person[2]$ have to be moved. Therefore, $Engine[1]$ and $Computer[1]$ will be moved to the entry of $Company[5]$. The child branch B_2 is affected when the entry of its parent branch is updated. Therefore, the information that associates with $Person[1]$ in $Course[1]$ will also be updated. Finally, the identity index for $Person[1]$, $Person[2]$, $Vehicle[1]$, $Engine[1]$ and $Computer[1]$ have to be updated. Additionally, if the attribute index is created for the classes involved for the moving, the attribute index will also be updated, for example, if an index is created for the *Name* attribute of the *Computer* class, this attribute index has to be updated by removing the associated addresses with corresponding to $Company[1]$ from the leaf node record and insert the address of the entry that corresponding to $Company[5]$ to that leaf node record.

B. Update the reference on different branches.

It is assumed that an object O in a class C of branch B_1 changes the reference from an object O' in a class C' of branch B_2 to an object O'' in the class C' of branch B_2 . Therefore, the identity index of the class C' of the branch B_2 has to be searched to find the entries that associate with the object O' and O'' and it is assumed that they are E_1 and E_2 respectively. The branch B_2 will be performed by deleting OID of O and the pointer of O from the entry of O' and insert them to the entry of O'' . For

example, $Person[1]$ of branch B_1 that previously references $Course[1]$ of branch B_2 is updated to reference $Course[2]$. Therefore, $Person[1]$ and its pointer from $Course[1]$ will be deleted and inserted to the entry of $Course[2]$. Finally, the identity index of $Person[1]$ and the attribute index involved on the branch B_2 will be updated.

5.4 Cost Model

In this section, the cost model in terms of storage overhead, retrieval cost and update cost will be formulated. The parameters that are used in the analysis will be given below.

Parameters:

- $N_{i,j}$: The number of objects in class j of branch i .
- $A_{i,j}$: The complex attribute of class j on branch i .
- $D_{i,j}$: Distinct value of complex attribute $A_{i,j}$.
- $UIDL$: The length of Object Identifier.
- P : Page size.
- pp : The size of page pointer.
- f : Average fan out from a non-leaf node.
- kl : Average length of a key value in attribute index.
- SL : The length of start field in the branch information.
- FSL : The length of free space in the branch information.
- PL : The length of pointer in branch information.
- $SA_{i,j,k}$: Simple attribute k of class j on branch i .
- $U_{i,j,k}$: The number of distinct values for simple attribute $SA_{i,j,k}$.
- $q_{i,j,k}$: The ratio of shared attribute value $= N_{i,j} / U_{i,j,k}$.
- Pk_i : Reference sharing of the parent class of class i .
- $k_{i,j}$: Reference sharing of class j on branch i .
- $nlb_{i,j}$: The number of leaf branch of class j on branch i .

Performance is measured by the number of I/O accesses. A page is used to estimate the storage overhead and the cost of performance because it is the

basic unit for data transfer between the main storage and the secondary storage. All lengths and sizes above are in bytes.

Assumptions:

1. There are no partial instantiation. This implies that $D_{i,j} = N_{i,j+1}$.
2. All key values have the same length.
3. All attributes are single-valued.

5.4.1 Storage Cost

In this subsection, the information of Figure 5.1 is used in the analysis. All branches are generated by using the algorithm of branch generation for the aggregation hierarchy of Figure 5.1.

Branch Information

For an aggregation hierarchy as a tree and after applying the algorithm of branch generation, It is assumed that m branches are generated.

Also, It is assumed that there are n classes in a branch B_i . These classes are related as in the form $C_1C_2C_3\dots C_n$. The size ($Size_{C_i}$) that associates with one object of the class C_i of B_i consists of the following:

- OID of this object for class C_i and its pointer to the child object.
- OIDs and pointers of the parent objects for the object in the first class of branch B_i .
- OIDs of leaf branch objects for class C_i of branch B_i .

$$Size_{C_i} = (UIDL + PL) + Pk_i * (UIDL + PL) + (nlb_{i,1}) * UIDL.$$

$$Size_{C_i} = (Pk_i + 1) * (UIDL + PL) + (nlb_{i,1}) * UIDL.$$

$$Size_{C_i} = (Pk_i + nlb_{i,1} + 1) * UIDL + (Pk_i + 1) * PL.$$

The size of an entry for one object in a class C_j of branch B_i ; when $2 \leq j \leq n - 1$; consists of the following:

- OID of this object for class C_j and its pointer to the child object.
- OIDs of leaf branch objects for class C_j of branch B_i .

$$Size_{C_j} = UIDL + PL + (nlb_{i,j}) * UIDL.$$

The size of an entry for one object in the class C_n of branch B_i :

$$Size_{C_n} = UIDL.$$

The size of an entry for every object in class C_1 of branch B_i :

$$SE(B_{i,1}) = \left(\prod_{j=1}^{n-1} k_{i,j} \right) * [(Pk_i + nlb_{i,1} + 1) * UIDL + (Pk_i + 1) * PL] + SL.$$

The size of an entry for every object in class C_j of branch B_i , $2 \leq j \leq n-1$:

$$SE(B_{i,j}) = \left(\prod_{l=j}^{n-1} k_{i,l} \right) * [(nlb_{i,j} + 1) * UIDL + PL] + SL.$$

The size of an entry for every object in class C_n of branch B_i :

$$SE(B_{i,n}) = UIDL + SL.$$

The total size of an entry of branch B_i :

$$SE(B_i) = SE(B_{i,1}) + \sum_{j=2}^{n-1} SE(B_{i,j}) + SE(B_{i,n}).$$

In case of the main branch B_j .

The size of an entry for every object in class C_j of branch B_1 , $1 \leq j \leq n-1$:

$$SE(B_{1,j}) = \left(\prod_{l=j}^{n-1} k_{1,l} \right) * [(nlb_{1,j} + 1) * UIDL + PL] + SL.$$

The size of an entry for every object in class C_n of branch B_1 :

$$SE(B_{1,n}) = UIDL + SL.$$

The total size of an entry of branch B_1 :

$$SE(B_1) = \sum_{j=1}^{n-1} SE(B_{1,j}) + SE(B_{1,n}).$$

If BP_i is the number of pages used for every entry in the branch B_i , then

$$BP_i = \begin{cases} \lceil N_{i,n} / \lfloor P / SE(B_i) \rfloor \rceil, & \text{if } SE(B_i) \leq P \\ N_{i,n} * \lceil SE(B_i) / P \rceil, & \text{if } SE(B_i) > P \end{cases}$$

The number of pages for the free space directory of branch B_i :

$$FSD_i = \lceil BP_i * (pp + FSL) / P \rceil.$$

The total size for all branches.

$$TBP = \sum_{i=1}^m (BP_i + FSD_i)$$

Identity Index

The identity index will be created for every object for each class of a branch. The average length of a leaf node index record for the identity index of class C_j .

$$XI = \begin{cases} UIDL + pp, & \text{if } C_j \text{ is not a leaf branch} \\ UIDL + Pk_j * pp. & \text{if } C_j \text{ is a leaf branch} \end{cases}$$

The number of leaf pages for the identity index of class C_j on branch B_i .

$$LP_{iden,i,j} = \lceil N_{i,j} / \lfloor P / XI \rfloor \rceil.$$

The number of non-leaf pages for the identity index of class C_j on branch B_i .

$$NLP_{iden,i,j} = \lceil LP_{iden,i,j} / f \rceil + \lceil \lceil LP_{iden,i,j} / f \rceil / f \rceil + \dots + x.$$

If $x < f$ and $x \neq 1$, 1 will be added in $NLP_{iden,i,j}$ for the root node. Therefore, the number of pages for the identity index of class C_j on branch B_i :

$$IIP_{i,j} = LP_{iden,i,j} + NLP_{iden,i,j}.$$

If there are n classes on the branch B_i , the number of pages for the identity index of branch B_i :

$$IIP_i = \sum_{j=1}^n IIP_{i,j}.$$

The number of of pages for the identity index of every branch is:

$$TIIP = \sum_{i=1}^m IIP_i.$$

Attribute Index

When creating the attribute index on a primitive value of a class j of branch B_j , $SA_{i,j,k}$ represents a primitive value k of the class j of branch B_j and an index is created on $SA_{i,j,k}$. The average length of a leaf node index record for the attribute index is:

$$XP_{SA_{i,j,k}} = kl + q_{i,j,k} * (UIDL + pp).$$

The number of leaf pages of the attribute index on branch B_j is:

$$LP_{SA_{i,j,k}} = \begin{cases} \lceil U_{i,j,k} / \lfloor P / XP_{SA_{i,j,k}} \rfloor \rceil, & \text{if } XP_{SA_{i,j,k}} \leq P \\ U_{i,j,k} * \lceil XP_{SA_{i,j,k}} / P \rceil, & \text{if } XP_{SA_{i,j,k}} > P \end{cases}$$

The number of non leaf pages of the attribute index on branch B_j is:

$$NLP_{SA_{i,j,k}} = \lceil LO_{SA_{i,j,k}} / f \rceil + \lceil \lceil LO_{SA_{i,j,k}} / f \rceil / f \rceil + \dots + x.$$

when $LO_{SA_{i,j,k}} = \min(U_{i,j,k}, LP_{SA_{i,j,k}})$ and $x < f$. If $x \neq 1$ add 1 to $NLP_{SA_{i,j,k}}$ for the root node.

Therefore, the number of pages for index on $SA_{i,j,k}$ is:

$$AIP_{SA_{i,j,k}} = LP_{SA_{i,j,k}} + NLP_{SA_{i,j,k}}.$$

Actually, many attribute indexes can be created. If there are n indexes on the branch B_j , the number of pages for these indexes is:

$$TAIP_i = \sum_{j=1}^n AIP_{i, index_j}$$

when $AIP_{i, index_j}$ is the j^{th} index of branch B_j .

The number of pages for the attribute index of every branch is:

$$TAIP = \sum_{i=1}^m TAIP_i.$$

Finally, the storage cost is:

$$SC = TBP + THP + TAIP.$$

5.4.2 Retrieval Cost

To simplify the analysis, It is assumed that there is only one predicate attribute in the queries and the predicate is specified on the indexed attribute. Cost formula will be classified as in the discussion in Section 5.3.1. Furthermore, the identity index is chosen to scan for the required entries instead of the forward traversal technique.

A. The predicate class and the target class are on the same branch

In this case, the predicate class and the target class are on the same branch. Therefore, it is convenient to perform the class traversal in the branch when using the Branch Index.

The retrieval cost of the branch index consists of the following:

- Cost of the attribute index scanning.
- Cost of the accessing the target objects from the target class for the qualified entries.

$$RC = h_{attr} + \lceil XP_{attr} / P \rceil + N_{P/Q} * \lceil SE_{B_i} / P \rceil.$$

when h_{attr} is the height of the attribute index-1, XP_{attr} is the length of a leaf node index record, $N_{P/Q}$ is the number of the qualified entries of a branch B_i for the predicate P of query Q .

B. The predicate and target class on different branches

In this case, the predicate class and the target class are on different branches for the Branch Index. The retrieval cost of the Branch Index can be classified on the location of the predicate class and the target class.

- *The predicate class is on an ancestor branch of the target class*

There is no information of the child branch stored in the parent branch. Therefore, after scanning the attribute index and obtain the qualified join objects from the join class, the identity index of the join class will be used to retrieve the qualified entries of the child branch. The general formula for the branch index when the predicate is on a branch j and the target class is on a branch k is:

$$RC = h_{attr} + \lceil XP_{attr} / P \rceil + N_{P/Q} * \lceil SE_{B_i} / P \rceil + \sum_{l=j}^{k-1} [N_{j,l+1} * (h_{iden} + 1) + N_{j,l+1} * \lceil SE_{B_{l+1}} / P \rceil].$$

when $N_{j,l+1}$ are the qualified objects of the join class that link between the branch l and the branch $l+1$ and there are several branches between the branch j and branch k .

- *The target class is on an ancestor branch of the predicate class*

Because the information of the child branch can link directly to its parent branch, the retrieval cost in this case is:

$$RC = h_{attr} + \lceil XP_{attr} / P \rceil + N_{P/Q} * \lceil SE_{B_i} / P \rceil + \sum_{l=j}^{k-1} N_{j,l+1} * \lceil SE_{B_{l+1}} / P \rceil.$$

5.4.3 Update Cost

The update cost will be formulated as discussed in Section 5.3.2. To simplify the analysis, The cost due to page overflow caused by update operation will not be included. When a complex attribute of one object is updated, the possible result is as follows.

- Update the reference on the same branch.
- Update the reference on different branches.

5.4.3.1 Update the Reference on the Same Branch

In this case, the update of the reference on the same branch of the branch index is considered.

Four different cases are categorized as follows.

A. The class of the updated object or its ancestor classes have no attribute index, no leaf branch and no child branch

In this case, It is assumed that the updated object is on the m^{th} class of branch i .

$$UC = 2 * (h_{iden} + 1 + 2 * \lceil SE_{B_i} / P \rceil) + \left(\sum_{l=1}^{m-1} \prod_{j=l}^{m-1} k_{i,j} + 1 \right) * (h_{iden} + 2).$$

when h_{iden} is the height of the identity index - 1.

B. The class of the updated object or its ancestor classes have an attribute index but no leaf branch and no child branch

$$UC = 2 * (h_{iden} + 1 + 2 * \lceil SE_{B_i} / P \rceil) + \left(\sum_{l=1}^{m-1} \prod_{j=l}^{m-1} k_{i,j} \right) * (h_{iden} + 2) + (h_{attr} + 2 \lceil XP_{attr} / P \rceil).$$

C. The class of the updated object or its ancestor classes have an attribute index and leaf branches but no child branch.

The number of objects for the leaf branches from objects of the first class to the updated objects is:

$$NLO = \left[\sum_{j=1}^{m-1} (nlb_{i,j} * \prod_{l=j}^{m-1} k_{i,l}) + nlb_{i,m} \right].$$

Therefore, the update cost is:

$$UC = 2 * (h_{iden} + 1 + 2 * \lceil SE_{B_i} / P \rceil) + \left(\sum_{l=1}^{m-1} \prod_{j=l}^{m-1} k_{i,j} + 1 + NLO \right) * (h_{iden} + 2) + (h_{attr} + 2 * \lceil XP_{attr} / P \rceil).$$

D. The class of the updated object or its ancestor classes have an attribute index, leaf branches and child branches.

Some parameters defined earlier will be used. So the update cost is:

$$UC = 2 * (h_{iden} + 1 + 2 * \lceil SE_{B_i} / P \rceil) + \left(\sum_{l=1}^{m-1} \prod_{j=l}^{m-1} k_{i,j} + 1 + NLO \right) * (h_{iden} + 2) + (h_{attr} + 2 * \lceil XP_{attr} / P \rceil) + \sum_{l=1}^m \left[\left(\prod_{j=1}^{m-l} k_{i,j} * ncb_{i,l} \right) * \left[(h_{iden} + 1) + 2 * \lceil SCB_{l,j} / P \rceil \right] \right].$$

when $ncb_{i,l}$ is the number of child branches of a branch i of class l and $SCB_{l,j}$ is the entry size of a child branch j of class l .

5.4.3.2 Update the Reference on Different Branches

The update of the branch index is performed only on the object of the first class of the child branch. The parent objects and associated pointers will be updated for the branch information of the child branch. Therefore, the update cost consists of the following:

- The scanning of the identity index for the old and new OID of object of the first class of the child branch.
- The update of the qualified entries of the child branch.
- The update of the identity index of the parent object.

$$UC = 2 * (h_{iden1} + 1 + 2 * \lceil SE_{B_j} / P \rceil) + (h_{iden2} + 2).$$

when SE_{B_j} is an entry size of a branch j ; the child branch of a branch i .

h_{iden1} is the height of the identity index - 1; of the first class of the branch j

h_{iden2} is the height of the identity index - 1; of the parent class of the branch j

The next chapter will compare all access methods presented in Chapter 3, Chapter 4 and Chapter 5 with the Path Dictionary Index of multi paths. The comparison of the storage cost, the retrieval cost and the update cost will be performed by assigning the value for the parameters and then analyzed them.

CHAPTER 6

COMPARISON OF ACCESS METHODS

This chapter analyzes the comparison of cost models between those of access methods presented in Chapter 3, Chapter 4, Chapter 5 and that of the Path Dictionary Index for multi paths. The condition of the reference sharing between objects is considered for all cases and the comparison is presented in the graphical form with the analysis.

6.1 Scope of the Comparison

The aggregation hierarchy as a tree in Figure 6.1 will be used for the analysis of cost models. The scope and various kinds of queries for the comparison of access methods will be defined in subsection.

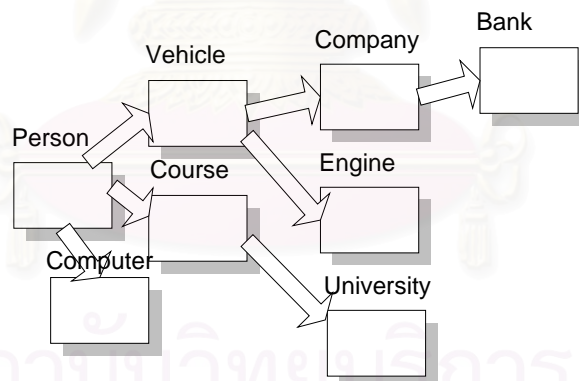


Figure 6.1 Aggregation hierarchy as a tree

6.1.1 The Number of Access Methods Used in Comparison

The access methods described in Chapter 3, Chapter 4 and Chapter 5 will be used in comparison because they are designed to cope with the aggregation hierarchy as a tree. Since the Path Dictionary Index is proved to be the best for the aggregation hierarchy in case of a path, multiple paths will be used to compare with the

three methods mentioned above. Therefore, the access methods used in comparison consist of the following.

1. Direct Access to Terminal Virtual Path.
2. Virtual path Signature.
3. Branch Index.
4. Path Dictionary Index

6.1.2 The Queries Used in Comparison.

It is assumed that the query is specified on the indexed attribute in case that the access method is one of the indexing techniques. The query will be categorized as follows.

- A. The predicate class is the root class and the target class is one of the remaining classes of the aggregation hierarchy as a tree.
 - The target class is an intermediate class.
 - The target class is a leaf class.

- B. The predicate class is a leaf class and the target class is one of the remaining classes of the aggregation hierarchy as a tree.
 - The target class is an intermediate class.
 - The target class is a leaf class.
 - The target class is the root class.

6.1.3 The parameters used in comparison

All parameters from Chapter 3, Chapter 4 and Chapter 5 will be used to analyze the cost models for the access methods. The chosen values of some parameters are adopted from [26] as they are listed in Table 6.1.

$UIDL$	= 8	$OFFL$	= 2	I	= 8
P	= 4096	SL	= 2	SZ	= 320
pp	= 4	FSL	= 2	S	= 2
f	= 218	EL	= 4	ll	= 2
kv	= 8	PL	= 2	cl	= 4
kl	= 1				

Table 6.1 Parameters of cost models

Performance is measured by the number of I/O accesses. A page is used to estimate the storage cost and the cost of performance because it is the basic unit for data transfer between the storage and the secondary storage. All lengths and sizes above are in bytes. To facilitate the cost models, these assumptions will be used as follows.

Assumptions:

1. All key values have the same length.
2. All attributes are single-valued.
3. All child objects are referenced by the parent objects

6.2 Storage Cost

The formula developed in Chapter 3, Chapter 4 and Chapter 5 will be used to compare the storage cost of these access methods with the path dictionary index. The attribute index will be created on one of the attributes of the *Person* class and

one of the attributes of the *University* class for the Branch Index and the Path Dictionary Index. The cardinality of the root class is fixed to 200,000.

Using the algorithm of branch generation for the aggregation hierarchy as a tree from Figure 6.1, two branches will be generated for the Branch Index. The main branch is from the *Person* class to the *Bank* class while the *Engine* class and the *Computer* class are its leaf branches. The second branch, the child branch, is from the *Course* class to the *University* class. Various paths of the Path Dictionary Index are created to mimic those of the Branch Index as follows.

Path 1: *Person* → *Vehicle* → *Company* → *Bank*

Path 2: *Vehicle* → *Engine*

Path 3: *Person* → *Course* → *University*

Path 4: *Person* → *Computer*

It is assumed that all reference sharing of all classes and the shared key values are set to the same value, which is represented as K . The impact of K to the storage overhead and cost of performance will be used to observe.

The storage cost of each access methods will be compared and mathematical proved by replacing the constant value of parameters from Table 6.1. The value of K that impacts the storage cost will be appeared in the formula for comparison.

- The Storage Cost of the Direct Access to Terminal Virtual Path.

$$SC = \lceil (N_1 * (nLC + 1) * UIDL) / P \rceil + NLP + LP.$$

$$SC = \frac{40N_1}{P} + \frac{19N_1}{KP} \left(1 + \frac{1}{f} + \frac{1}{f^2} + \dots \right).$$

Since the value of f is high, so the storage cost is:

$$SC = \frac{N_1}{P} \left(40 + \frac{19}{K} \right).$$

- The Storage Cost of the Virtual Path Signature.

$$SC = \lceil (N_1 * (2S + UIDL * (nLC + nNLC))) / P \rceil.$$

$$SC = \frac{N_1}{P} (68).$$

- The Storage cost of the Branch Index.

Branch Information

$$SE(B_1) = 18K^3 + 18K^2 + 10K + 16.$$

$$SE(B_2) = 10K^2 + 10K + 2.$$

$$BP_1 = \frac{N_1}{P} \left(18 + \frac{18}{K} + \frac{10}{K^2} + \frac{16}{K^3} \right).$$

$$BP_2 = \frac{N_1}{P} \left(10 + \frac{10}{K} + \frac{2}{K^2} \right).$$

The term of *SFD* is ignored because it is very small when compared with *BP*. Therefore, the total branch information is:

$$TBP = \frac{N_1}{P} \left(28 + \frac{28}{K} + \frac{12}{K^2} + \frac{16}{K^3} \right).$$

Attribute Index

$$XP = 1 + 12K.$$

$$LP = \frac{N_1}{P} \left(\frac{1}{K} + 12 \right).$$

NLP is very small when compared with *LP*. Therefore, the attribute index is:

$$AI_{Person} = \frac{N_1}{P} \left(\frac{1}{K} + 12 \right).$$

$$AI_{University} = \frac{N_1}{P} \left(\frac{1}{K^3} + \frac{12}{K^2} \right).$$

$$TAIP = \frac{N_1}{P} \left(\frac{1}{K^3} + \frac{12}{K^2} + \frac{1}{K} + 12 \right).$$

Identity Index

$$LP_{Bank} = \frac{N_1}{P} \left(\frac{12}{K^3} \right). \quad LP_{Company} = \frac{N_1}{P} \left(\frac{12}{K^2} \right). \quad LP_{Vehicle} = \frac{N_1}{P} \left(\frac{12}{K} \right).$$

$$LP_{Person} = \frac{N_1}{P} (12). \quad LP_{University} = \frac{N_1}{P} \left(\frac{12}{K^2} \right). \quad LP_{Course} = \frac{N_1}{P} \left(\frac{12}{K} \right).$$

$$LP_{Engine} = \frac{N_1}{P} \left(\frac{8}{K^2} + \frac{4}{K} \right). \quad LP_{Computer} = \frac{N_1}{P} \left(\frac{8}{K} + 4 \right).$$

$$TIIP = \frac{N_1}{P} \left(28 + \frac{36}{K} + \frac{32}{K^2} + \frac{12}{K^3} \right).$$

Therefore, the storage cost of the Branch Index is:

$$SC = TBP + TAIP + TIIP.$$

$$SC = \frac{N_1}{P} \left(68 + \frac{65}{K} + \frac{56}{K^2} + \frac{29}{K^3} \right).$$

- The Storage cost of the Path Dictionary Index.

S-Expression

$$SS(P_1) = 10K^3 + 10K^2 + 10K + 10. \quad SS(P_2) = 10K + 16.$$

$$SS(P_3) = 10K^2 + 10K + 18. \quad SS(P_4) = 10K + 16.$$

$$SSP_1 = \frac{N_1}{P} \left(10 + \frac{10}{K} + \frac{10}{K^2} + \frac{10}{K^3} \right). \quad SSP_2 = \frac{N_1}{P} \left(\frac{10}{K} + \frac{16}{K^2} \right).$$

$$SSP_3 = \frac{N_1}{P} \left(10 + \frac{10}{K} + \frac{18}{K^2} \right). \quad SSP_4 = \frac{N_1}{P} \left(10 + \frac{16}{K} \right).$$

The total s-expression is:

$$TSS = \frac{N_1}{P} \left(30 + \frac{46}{K} + \frac{44}{K^2} + \frac{10}{K^3} \right).$$

Attribute Index

$$XP = 1 + 12K.$$

$$LP = \frac{N_1}{P} \left(\frac{1}{K} + 12 \right).$$

NLP is very small when compared with LP . Therefore, the attribute index is:

$$AI_{Person} = \frac{N_1}{P} \left(\frac{1}{K} + 12 \right).$$

$$AI_{University} = \frac{N_1}{P} \left(\frac{1}{K^3} + \frac{12}{K^2} \right).$$

$$TAIP = \frac{N_1}{P} \left(\frac{1}{K^3} + \frac{12}{K^2} + \frac{1}{K} + 12 \right).$$

Identity Index

The identity index is created for every object of all classes in the path.

$$LP_1 = \frac{N_1}{P} \left(\frac{12}{K^3} + \frac{12}{K^2} + \frac{12}{K} + 12 \right). \quad LP_2 = \frac{N_1}{P} \left(\frac{12}{K^2} + \frac{12}{K} \right).$$

$$LP_3 = \frac{N_1}{P} \left(\frac{12}{K^2} + \frac{12}{K} + 12 \right). \quad LP_4 = \frac{N_1}{P} \left(\frac{12}{K} + 12 \right).$$

$$TIIP = \frac{N_1}{P} \left(36 + \frac{48}{K} + \frac{36}{K^2} + \frac{12}{K^3} \right).$$

Therefore, the storage cost of the Path Dictionary Index is:

$$SC = TSS + TAIP + TIIP.$$

$$SC = \frac{N_1}{P} \left(78 + \frac{95}{K} + \frac{92}{K^2} + \frac{23}{K^3} \right).$$

The upper bound and the lower bound of each access methods can be obtained by considering the value of K . The lower bound occurs when the value of K is high and the upper bound occurs when the value of K is equal to 1. Therefore the boundary of the storage cost of each access method is as follows.

$$\frac{40N_1}{P} \leq SC_{DTVP} \leq \frac{59N_1}{P}.$$

$$SC_{VPS} = \frac{68 N_1}{P}.$$

$$\frac{68N_1}{P} \leq SC_{BI} \leq \frac{218N_1}{P}.$$

$$\frac{78N_1}{P} \leq SC_{PDI} \leq \frac{288N_1}{P}.$$

It is proved that the storage cost of the Direct Access to Terminal Virtual Path is the lowest. The storage cost of the Virtual Path Signature is constant and it is the second lowest. Although, the storage cost of the Branch Index is lower than that of the Path Dictionary Index, it is never less than those of the Direct Access to Terminal Virtual Path and the Virtual Path Signature for all ranges of K .

The analysis by using graphical view is shown in Figure 6.2 by varying the value of K from 2 to 10 at the x-axis.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

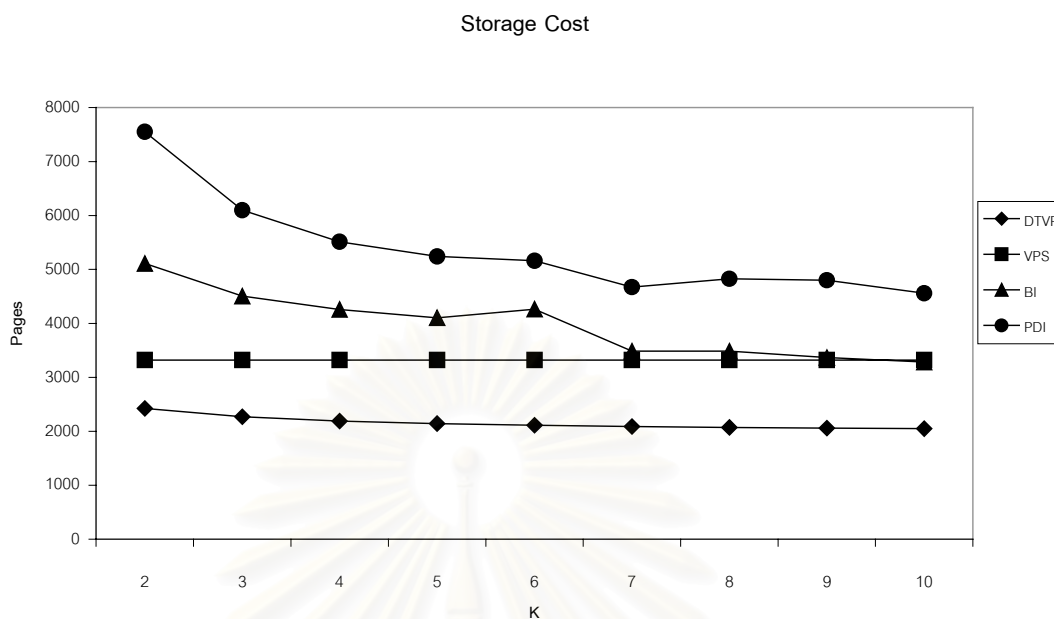


Figure 6.2 The storage cost of access methods

- *Comparison:*

From Figure 6.2, the Direct Access to Terminal Virtual Path (DTVP) has the lowest storage cost. Actually, the cost of the linking file structure is constant but the cost of the attribute index of the root class is varied from the value of K . When K is bigger, the cost of the attribute index is lower. The result why DTVP has the lowest storage is due to its structure. As mentioned in Chapter 3, the structure of the linking file structure is stored by OID of the root class and associated OIDs of the leaf classes in Terminal Virtual Path. Therefore, less storage is required when the OIDs of the intermediate classes are ignored.

The storage cost of Virtual Path Signature (VPS) is constant and it is the second lowest. As mentioned from Chapter 4, the structure of the signature file depends on the signature of the Virtual Path and the number of classes in the aggregation hierarchy as a tree. Since the cardinality of the *Person* class is constant and the number of entries of the signature file is equal to the number of objects in the root class, so the storage cost of the Virtual Path Signature is constant and it does not depend on K .

The storage cost of the Virtual Path Signature is higher than that of the Direct Access to Terminal Virtual Path because of the additional cost from the Non-Terminal Virtual Path.

It is noticeable from Figure 6.2 that the storage cost of the Branch Index is lower than that of the Path Dictionary Index. The result for the less storage is as follows.

- The total cost of branch information for two branches is lower than the total cost of s-expressions for four paths.
- The overall identity index of branch index is lower.

6.3 Retrieval Cost

The same parameters will be used as in the case of the analysis of the storage cost. To simplify the analysis, It is assumed that there is only one predicate attribute in the query and the predicate is specified on the indexed attribute. As described in Chapter 3, the Direct Access to Terminal Virtual Path (DTVP) is applicable when the predicate is specified on the indexed attribute of the root class and the target class can be any leaf classes of the aggregation hierarchy as a tree. Therefore, there will be only three access methods, i.e. the Virtual Path Signature (VPS), the Branch Index (BI) and the Path Dictionary Index (PDI) for the comparison of the retrieval cost when the condition of the query does not cover the Direct Access to Terminal Virtual Path (DTVP). The comparison will be performed as mentioned in subsection 6.1.2.

6.3.1 The Predicate Class is the Root Class.

In this case, the predicate is specified on the indexed attribute of the *Person* class and the target class is one of the remaining classes.

- The target class is an intermediate class.

From Figure 6.1, the intermediate classes are the *Vehicle* class, the *Company* class and the *Course* class. The comparison of the retrieval cost will be performed only for the Virtual Path Signature (VPS), the Branch Index (BI) and the Path Dictionary Index (PDI).

- The Retrieval Cost of the Virtual Path Signature.

$$RC = \frac{N_1}{P}(68) + R * N_1 * \lceil 320 / P \rceil.$$

- The Retrieval Cost of the Branch Index.

$$RC = h_{attr} + 1 + K * \lceil SE_{B_1} / P \rceil.$$

$$\text{When } SE_{B_1} = 18K^3 + 18K^2 + 10K + 16.$$

- The Retrieval Cost of the Path Dictionary Index.

$$RC = h_{attr} + 1 + K * \lceil SS_{P_1} / P \rceil.$$

$$\text{When } SS_{P_1} = 10K^3 + 10K^2 + 10K + 10.$$

It is proved that the size of SE_{B_1} grows more quickly than that of SS_{P_1} . Therefore, it is concluded as follows.

$$SE_{B_1} > P \text{ when } K \geq 6.$$

$$SS_{P_1} > P \text{ when } K \geq 8.$$

The retrieval cost of the Virtual Path Signature is constant and it depends on the number of objects of the root class. Since the value of N_1 is much bigger than the value of K in SE_{B_1} or SS_{P_1} , the retrieval cost of the Virtual Path Signature is the highest.

The retrieval cost of the Branch Index is equal to that of the Path Dictionary Index when SE_{B_1} is equal to SS_{P_1} , i.e. when $K < 6$. However, when $K \geq 6$, the size of SE_{B_1} grows more quickly than that of SS_{P_1} . Therefore, it is proved that

the retrieval cost of the Branch Index is more than that of the Path Dictionary Index when $K \geq 6$.

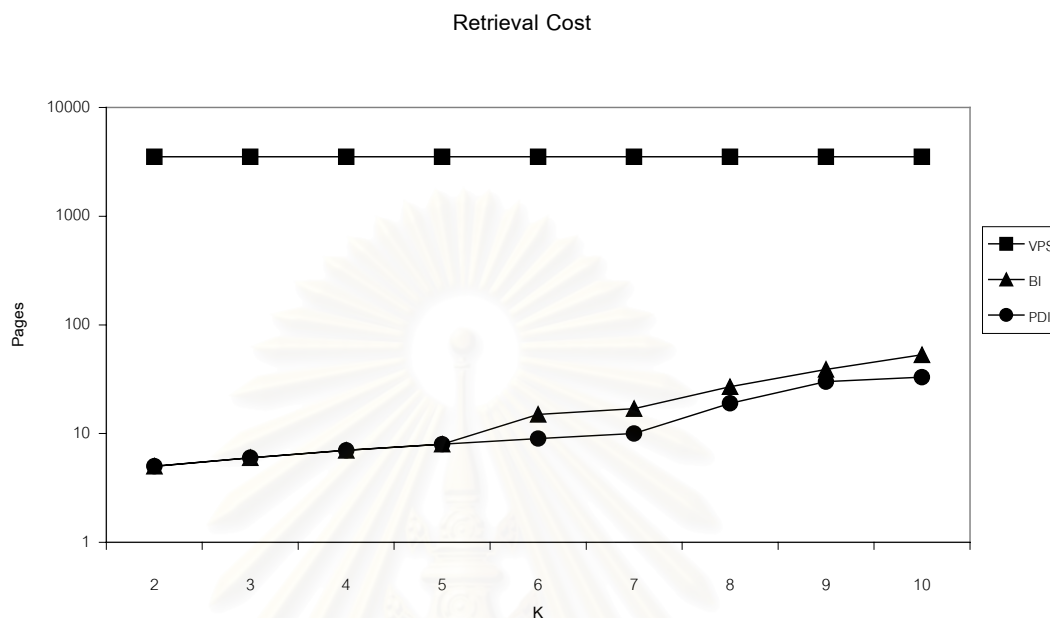


Figure 6.3 The retrieval cost when the predicate class is the *Person* class and the target class is the *Vehicle* class

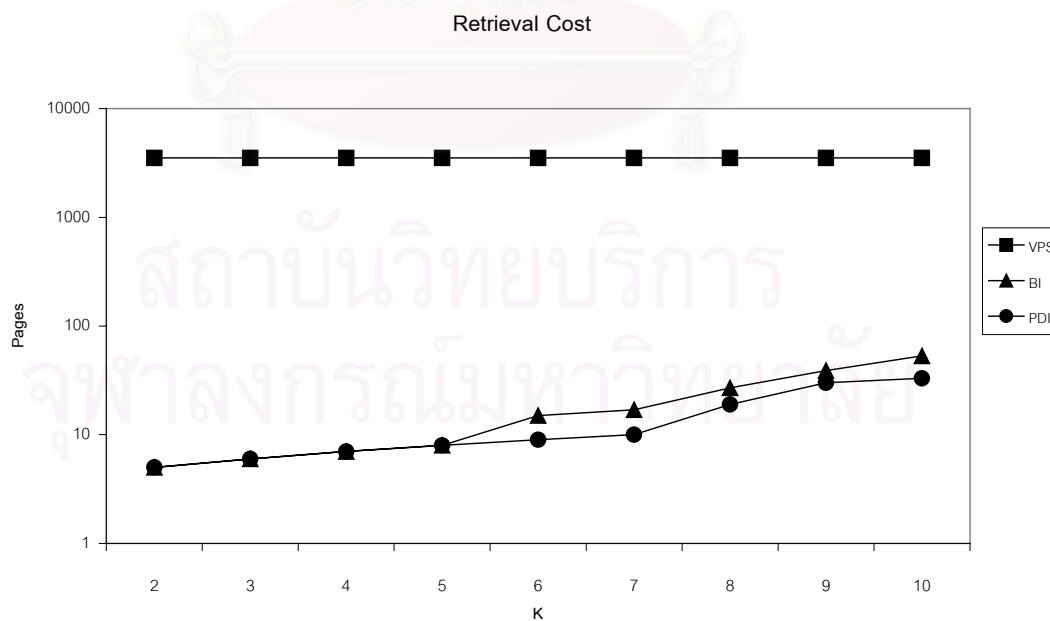


Figure 6.4 The retrieval cost when the predicate class is the *Person* class and the target class is the *Company* class

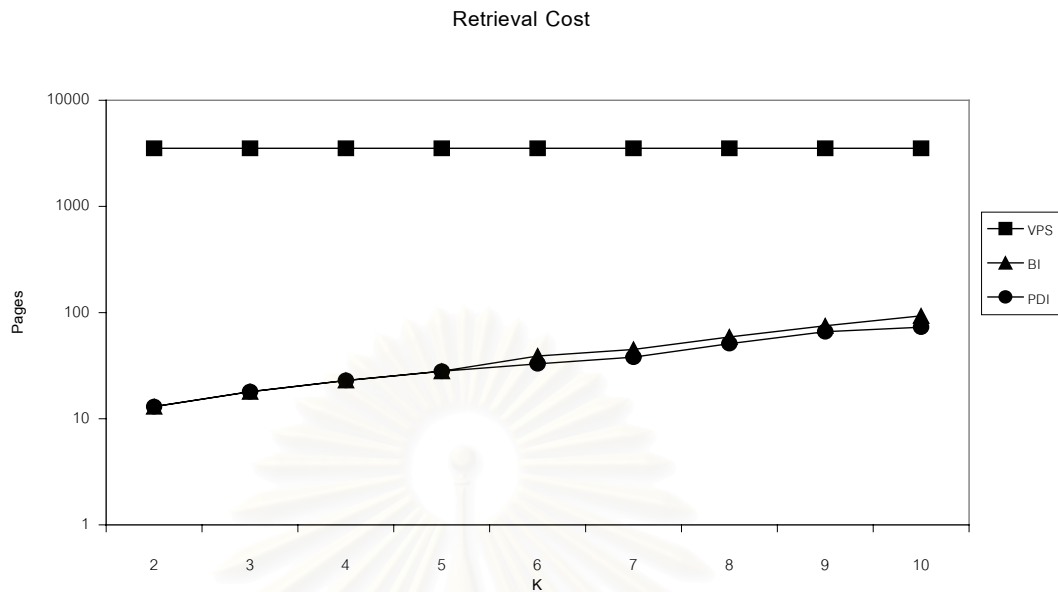


Figure 6.5 The retrieval cost when the predicate class is the *Person* class and the target class is the *Course* class.

- Comparison:

The retrieval cost of the Virtual Path Signature (VPS) from Figure 6.3, Figure 6.4 and Figure 6.5 is constant. It consists of scanning all entries of the signature file and for the qualified entries, accessing the candidate objects from the *Person* class to resolve the false drop. Since most of the retrieval cost of the Virtual Path Signature comes from the scanning of the signature file, so the retrieval cost of the Virtual Path Signature is much higher than the other two access methods. The retrieval cost of the Branch Index (BI) and the Path Dictionary Index (PDI) consists of scanning the attribute index and retrieve the qualified branch information or s-expression from the Branch Index and the Path Dictionary Index respectively. It is noticeable from these figures that the retrieval cost of the Branch Index is a little higher than that of the Path Dictionary Index because there are more information in the main branch of the Branch Index than in the Path1' s-expression of the Path Dictionary Index. However, the retrieval cost of the Branch Index and Path Dictionary Index is the same when K is less than 6, because the storage of the branch information and the s-expression are still less than a page size.

- The target class is a leaf class.

The comparison of the retrieval cost will be performed for the Direct Access to Terminal Virtual Path (DTVP), the Virtual Path Signature (VPS), the Branch Index (BI) and the Path Dictionary Index (PDI). Four leaf classes in Figure 6.1; i.e. the *Bank* class, the *Engine* class, the *University* class, the *Computer* class; will be considered as the target class respectively.

When the target is at the leaf branch class such as the *Engine* class or the *Computer* class, the retrieval cost of access methods is:

- The Retrieval Cost of the Direct Access to Terminal Virtual Path.

$$RC = h_{attr} + 1 + K * \lceil 40K / P \rceil.$$

- The Retrieval Cost of the Virtual Path Signature.

$$RC = \frac{N_1}{P}(68) + R * N_1 * \lceil 320 / P \rceil.$$

- The Retrieval Cost of the Branch Index.

$$RC = h_{attr} + 1 + K * \lceil SE_{B_1} / P \rceil.$$

$$\text{When } SE_{B_1} = 18K^3 + 18K^2 + 10K + 16.$$

- The Retrieval Cost of the Path Dictionary Index.

$$RC = h_{attr} + 1 + K * \lceil SS_{P_1} / P \rceil + K((h_{iden} + 1) + \lceil SS_{P_i} / P \rceil).$$

$$\text{When } SS_{P_1} = 10K^3 + 10K^2 + 10K + 10 \text{ and } SS_{P_i} = 10K + 16.$$

It is proved that the retrieval cost of the Direct Access to Terminal Virtual path is the lowest because the value of the last term in the equation is much lower than those of the remaining access methods. The retrieval cost of the Path Dictionary Index is higher than that of the Branch Index because of the additional terms of the equation. However, the retrieval cost of the Branch Index will close to that of the Path Dictionary

Index when the value of K is high because the value of SE_{B_1} grows more quickly than that of SS_{P_1} and SS_{P_i} .

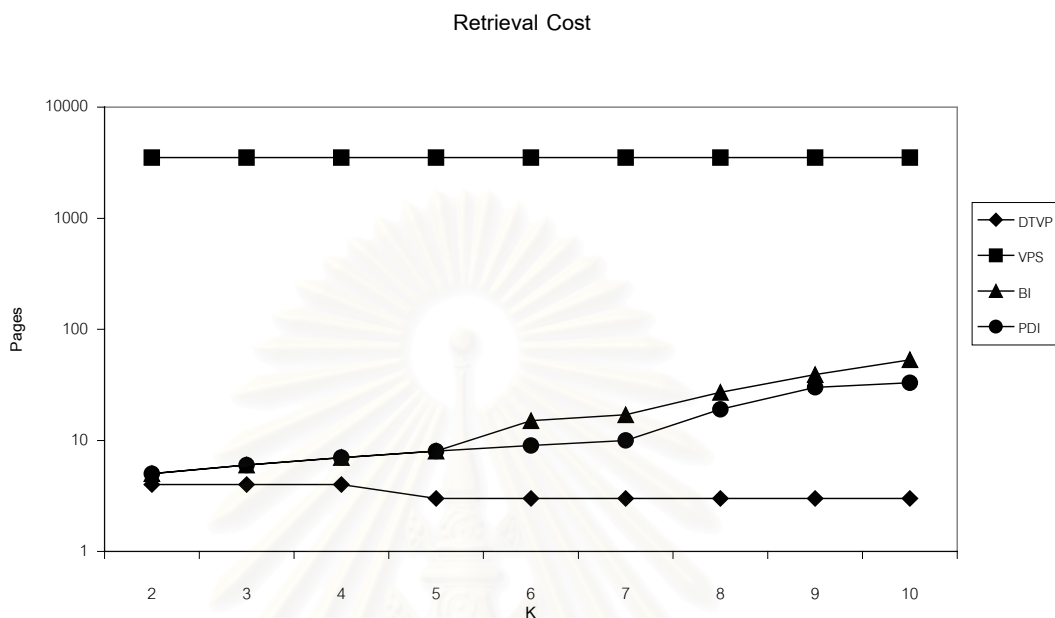


Figure 6.6 The retrieval cost when the predicate class is the *Person* class and the target class is the *Bank* class

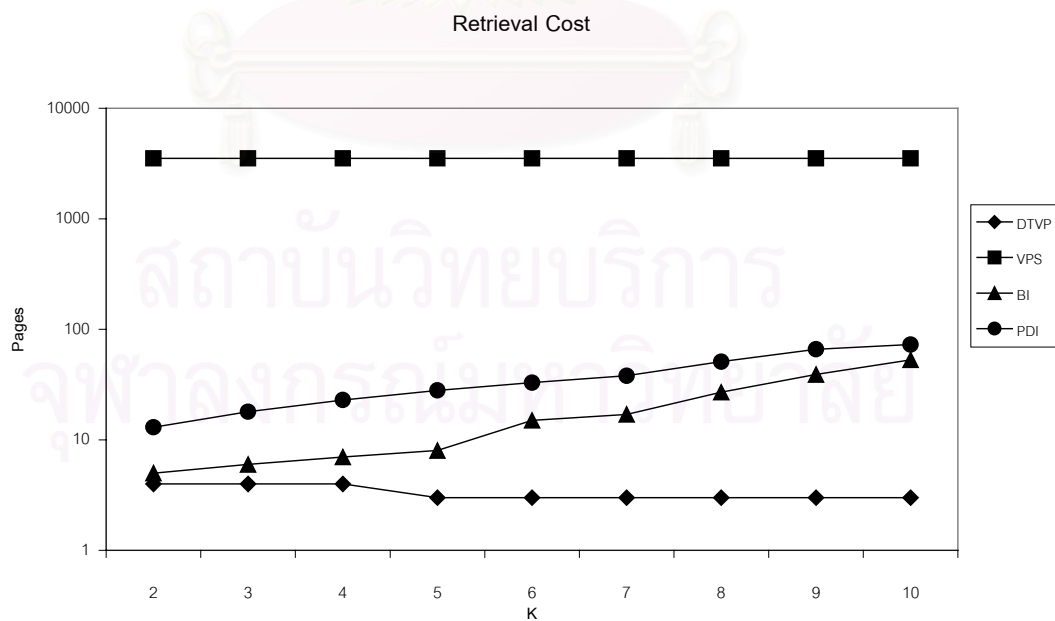


Figure 6.7 The retrieval cost when the predicate class is the *Person* class and the target class is the *Engine* class

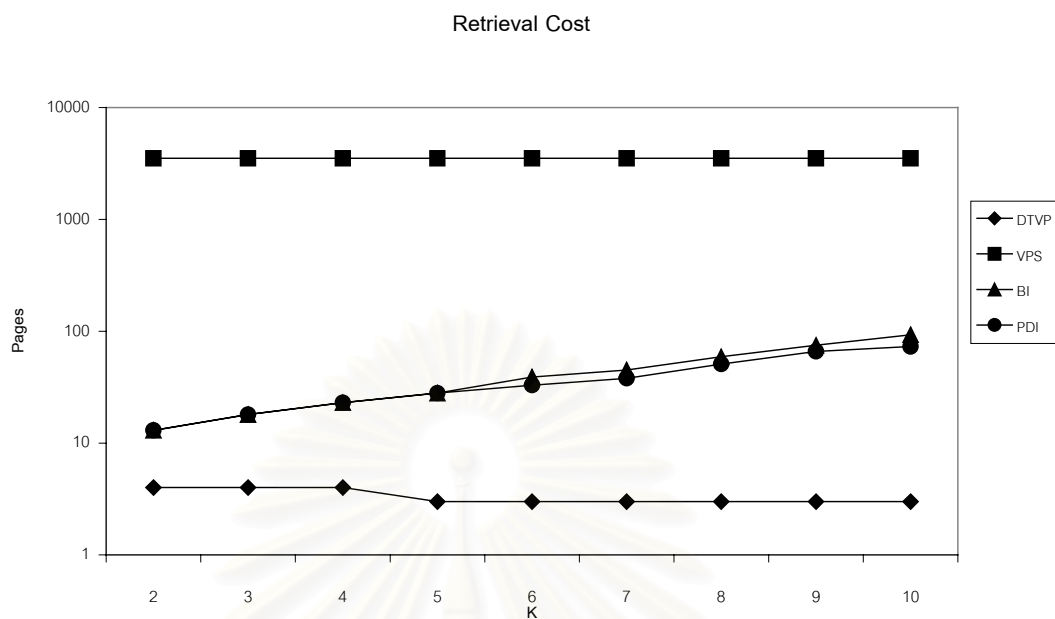


Figure 6.8 The retrieval cost when the predicate class is the *Person* class and the target class is the *University* class

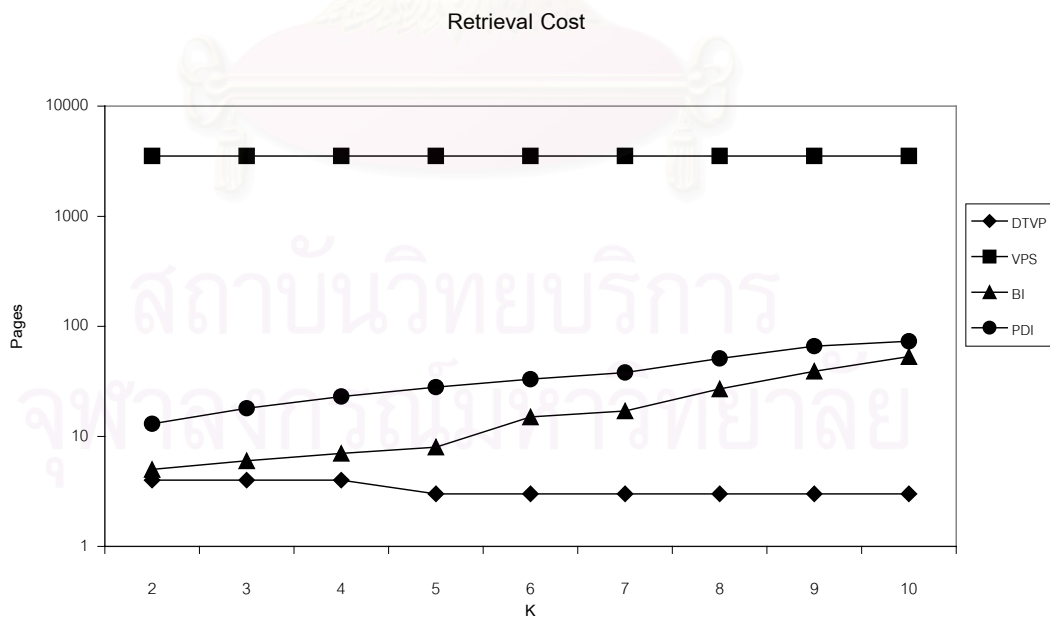


Figure 6.9 The retrieval cost when the predicate class is the *Person* class and the target class is the *Computer* class

- *Comparison:*

From Figure 6.6, Figure 6.7, Figure 6.8 and Figure 6.9, the retrieval cost of the Direct Access to Terminal Virtual Path (DTVP) is the best. As described from Chapter 3, the information stored in an entry of the linking file structure is OID of the Person class and associated OIDs of the leaf classes in the Terminal Virtual Path (TVP). Since all entries of the linking file structure have the same space and they are less than a page, retrieval of the qualified entries after obtaining the attribute index is done efficiently. When the key shared value is more than 4, the retrieval cost is lower because the leaf node record is reduced until there is only one root node left.

The retrieval cost of the Virtual Path Signature (VPS) is the same as in the case that the target class is an intermediate class. It is noticeable that the location of the target class and value of K do not affect its retrieval cost. However, if we do not know what attribute will be involved in the predicate and that attribute is not indexed, the signature technique would be an alternative approach for accessing the qualified objects.

As the target class is the *Engine* class and the *Computer* class from Figure 6.7 and Figure 6.9, the retrieval cost of the Branch Index (BI) is lower than that of the Path Dictionary Index because the *Engine* class and the *Computer* class are leaf branches of the main branch. However, for the Path Dictionary Index, the *Engine* class and the *Computer* class are on other path dictionaries so that more accesses are required. The retrieval cost of the Branch Index is a little higher than that of the Path Dictionary Index in Figure 6.6 and Figure 6.8 when K is greater than 5. Since the *Person* class and *Bank* class are in the same branch of the Branch Index and they are also in the same path of the Path Dictionary Index, the Path Dictionary Index will gain a little lower retrieval cost. As the target class is the *University* class, the child branch of the Branch Index is required for obtaining the target objects so that a littler higher cost is occurred.

6.3.2 The Predicate Class is a Leaf Class.

In this case, the predicate is specified on the indexed attribute of the *University* class and the target class is one of the remaining classes. The mathematical proof can be performed as the same way as in Section 6.3.1.

- The target class is an intermediate class.

The intermediate classes are the *Vehicle* class, the *Company* class and the *Course* class. The comparison of the retrieval cost will be performed only for the Virtual Path Signature (VPS), the Branch Index (BI) and the Path Dictionary Index (PDI).

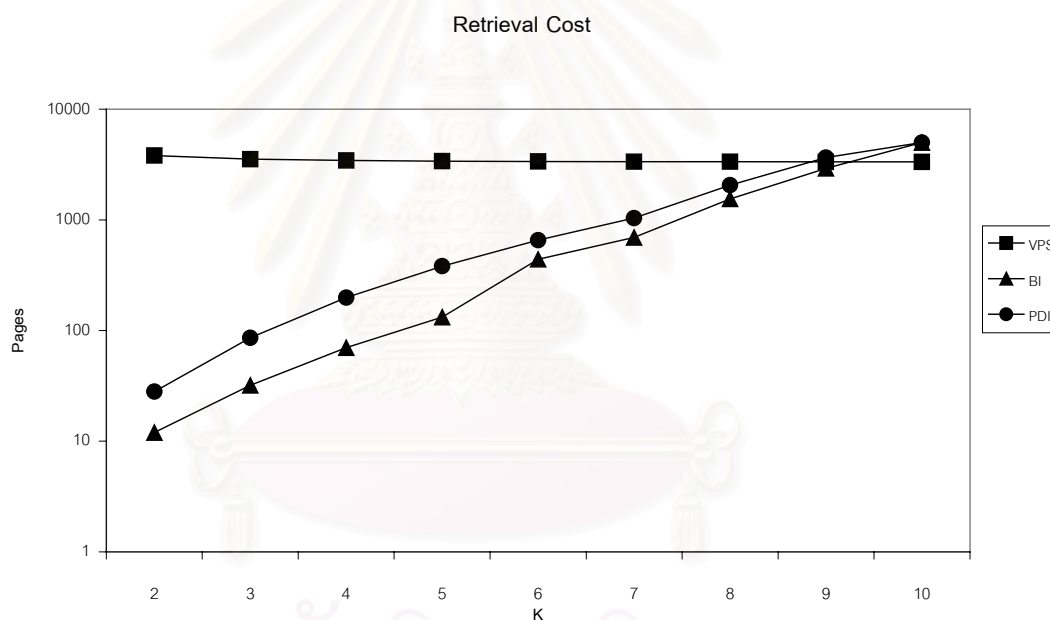


Figure 6.10 The retrieval cost when the predicate class is the *University* class and the target class is the *Vehicle* class

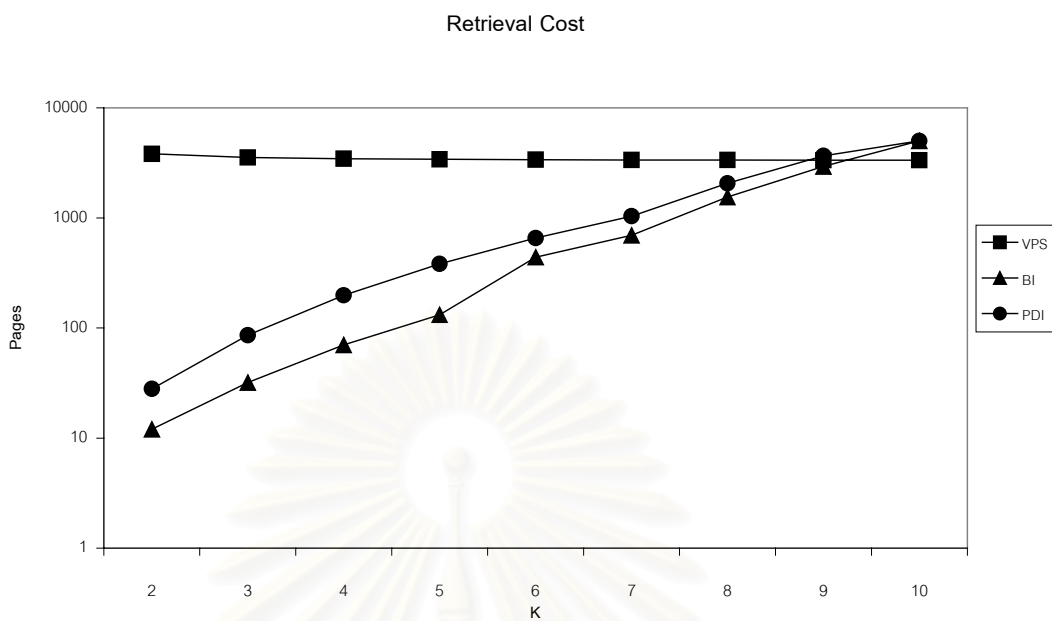


Figure 6.11 The retrieval cost when the predicate class is the *University* class and the target class is the *Company* class

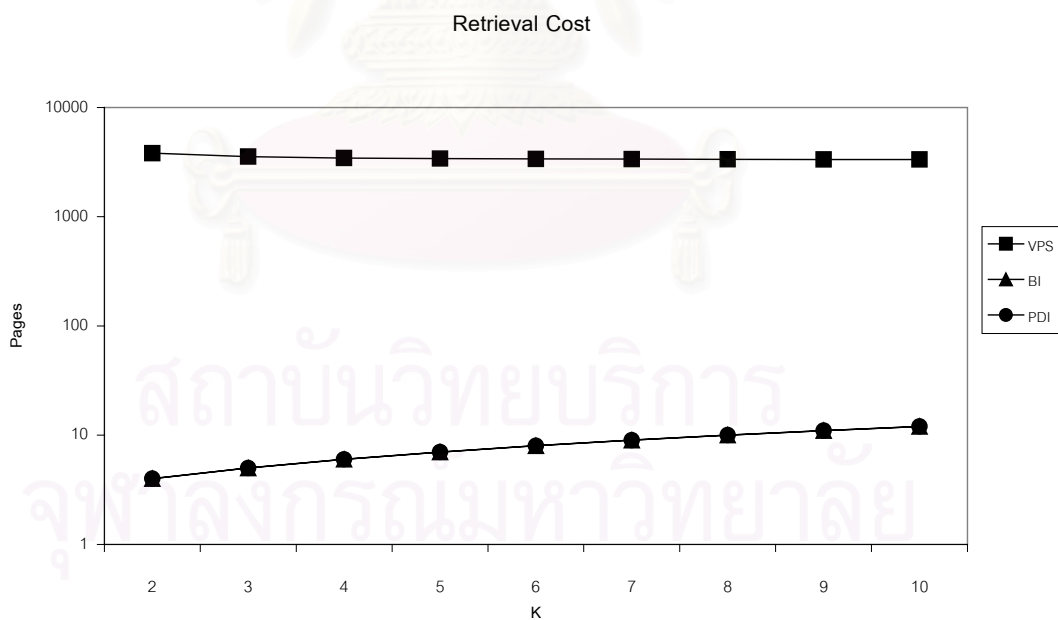


Figure 6.12 The retrieval cost when the predicate class is the *University* class and the target class is the *Course* class

- Comparison:

From Figure 6.10, Figure 6.11 and Figure 6.12, the retrieval cost of the Virtual Path Signature (VPS) has tendency to lower when K is higher because the number of objects in the *University* class is lower when K is higher and so do the candidate objects. However, its retrieval cost is still high when compared with those of the Branch Index (BI) and the Path Dictionary Index (PDI).

It is shown clearly that the retrieval cost of the Branch Index in Figure 6.10 and Figure 6.11 is lower than that of the Path Dictionary Index. As described from Chapter 5, the child branch of the Branch Index can link to its parent branch so that cost of object traversal is reduced. However, for the Path Dictionary Index, Path 3 has to be considered first to obtain the qualified objects in the Person class. Then Path 1 has to be searched next by using the identity index of the qualified objects from Path 3.

From Figure 6.12, the retrieval cost of the Branch Index and that of the Path Dictionary is the same because the *Course* class and the *University* class are on the same branch information and s-expression respectively. Furthermore, the storage of each branch information and s-expression is not more than a page size so that only one page is required to retrieve the branch information or the s-expression.

- The target class is a leaf class.

The comparison of the retrieval cost will be performed for the Virtual Path Signature (VPS), the Branch Index (BI) and the Path Dictionary Index (PDI). The three remaining leaf classes in Figure 6.1; i.e. the *Bank* class, the *Engine* class, the *Computer* class; will be considered as the target class respectively.

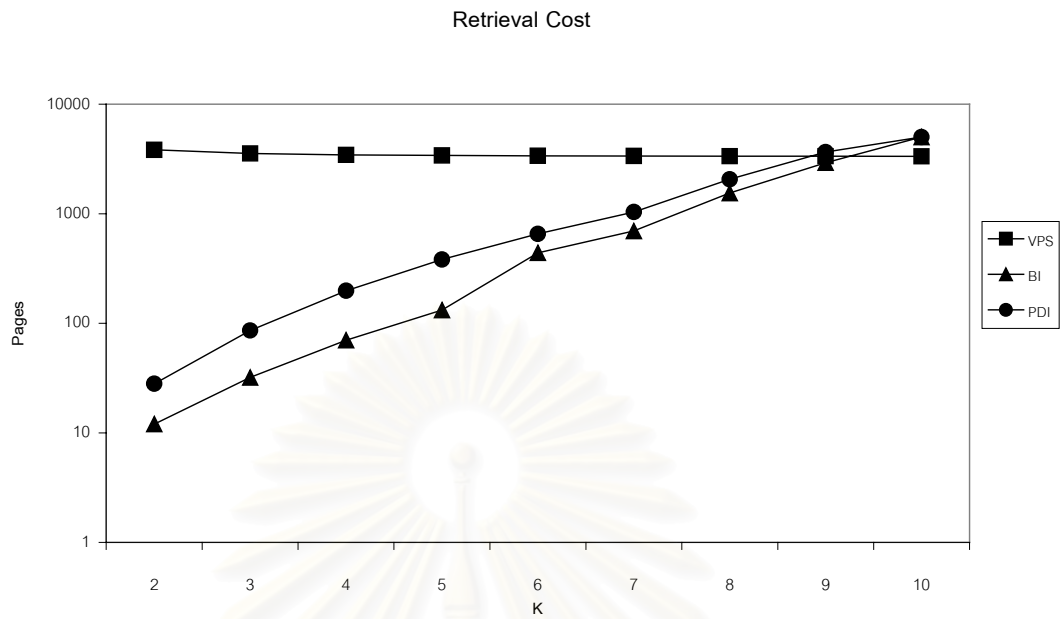


Figure 6.13 The retrieval cost when the predicate class is the *University* class and the target class is the *Bank* class

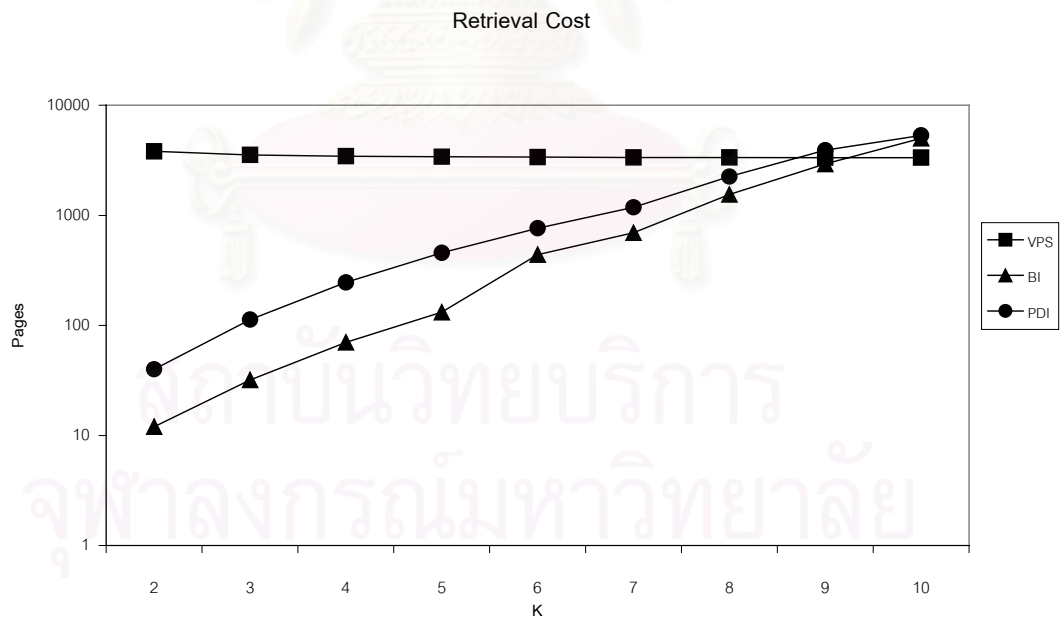


Figure 6.14 The retrieval cost when the predicate class is the *University* class and the target class is the *Engine* class

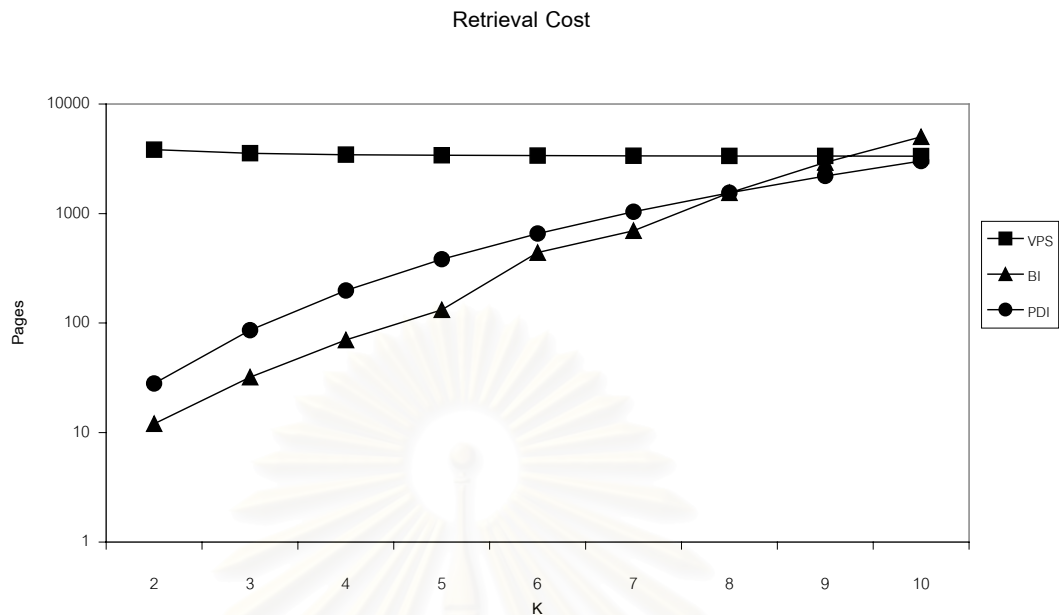


Figure 6.15 The retrieval cost when the predicate class is the *University* class and the target class is the *Computer* class

- Comparison:

From Figure 6.13, Figure 6.14 and Figure 6.15, it is shown that the retrieval cost of the Branch Index (BI) is lower than that of the Path Dictionary Index (PDI). As explained previously, the *Bank* class, the *Engine* class and the *Computer* class are on the same main branch but they are on different paths of the Path Dictionary Index. It is noticeable that the retrieval cost of the Branch Index and that of the Path Dictionary Index will close to each other when the value of K increases. That is because more objects of the *Person* class are required so that the accessing of the branch information compensate the traversal of multiple paths of the Path Dictionary Index.

- The target class is the root class.

In this case, the *Person* class will be considered as the target class.

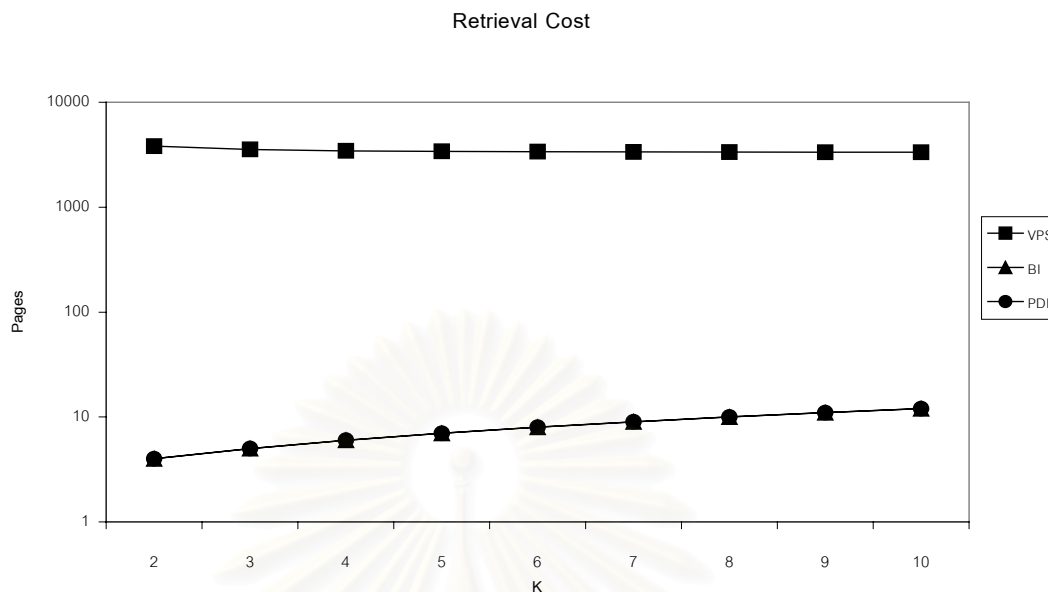


Figure 6.16 The retrieval cost when the predicate class is the *University* class and the target class is the *Person* class

- Comparison:

The retrieval cost of the Branch Index (BI) and that of the Path Dictionary Index is the same because the *Person* class and the *University* class are on the same branch information and s-expression as explained earlier.

6.4 Update Cost

Update cost analysis will be considered only for the Branch Index and the Path Dictionary Index because they are efficient for retrieval and they covers all queries as analyzed in Section 6.3. Since update simple attributes does not affect the data in the branch information or the s-expression, only update of the object reference between classes of the aggregation hierarchy of Figure 6.1 will be considered. The same parameters will be used to analyze as in the case of the storage cost and the retrieval cost.

- Update the Reference of Class in the Same Branch.

In this case, the update of reference between the *Vehicle* class and the *Company* class is used for the mathematical proof.

- Update Cost of the Branch Index.

$$UC = 2 * (h_{iden} + 1 + 2 * \lceil (18K^3 + 18K^2 + 10K + 16) / P \rceil) + 2 * (1 + K)(h_{iden} + 2) \\ + (h_{attr} + 2 \lceil XP / P \rceil) + UC_{CB} .$$

When UC_{CB} is the update cost of the child branch.

- Update Cost of the Path Dictionary Index.

$$UC = 2 * (h_{iden} + 1 + 2 * \lceil (10K^3 + 10K^2 + 10K + 10) / P \rceil) + (1 + K)(h_{iden} + 2) \\ + (h_{attr} + 2 \lceil XP / P \rceil) .$$

Since there are more terms in the formula of the update cost of the Branch Index, especially the term of the update of the child branch, the update cost of the Branch Index is proved to be higher than that of the Path Dictionary Index.

However, If update is performed to the leaf branch such as the reference between the *Vehicle* class and the *Engine* class, the update cost of the Branch Index is as follow.

$$UC = 2 * (h_{iden} + 1 + 2 * \lceil (18K^3 + 18K^2 + 10K + 16) / P \rceil) + (h_{iden} + 2) .$$

It is proved that the update cost of the attribute index of the parent class is unnecessary. Therefore when $K < 6$, the update cost of the Branch Index will lower than that of the Path Dictionary Index. However, as $K \geq 6$ the benefit gains is not much enough to compensate the higher value of SE_{B_1} .

- Update the Reference of Class in Different Branches.

In this case, the update is performed between the *Person* class and the *Course* class.

- Update Cost of the Branch Index.

$$UC = 2 * (h_{iden1} + 1 + 2 * \lceil SE_{B2} / P \rceil) + (h_{iden2} + 2).$$

$$\text{When } SE_{B2} = 10K^2 + 10K + 2,$$

h_{iden1} is the height of the identity index of the *Course* class,

h_{iden2} is the height of the identity index of the *Person* class.

- Update Cost of the Path Dictionary Index.

$$UC = 2 * (h_{iden} + 1 + 2 * \lceil SS_{P3} / P \rceil) + (h_{iden} + 2).$$

$$\text{When } SS_{P3} = 10K^2 + 10K + 18,$$

h_{iden} is the height of the identity index of the Path Dictionary *P3*.

As mentioned from the storage cost, the leaf page of the identity index of the *Course* class is $\frac{N_1}{P} \left(\frac{12}{K} \right)$ and the leaf page of the identity index of the *Person* class is $\frac{N_1}{P} (12)$, but the leaf page of the identity index of the Path Dictionary *P3* is $\frac{N_1}{P} \left(12 + \frac{12}{K} + \frac{12}{K^2} \right)$. It is proved that the update cost of the Path Dictionary Index in this case never lower than that of the Branch Index.

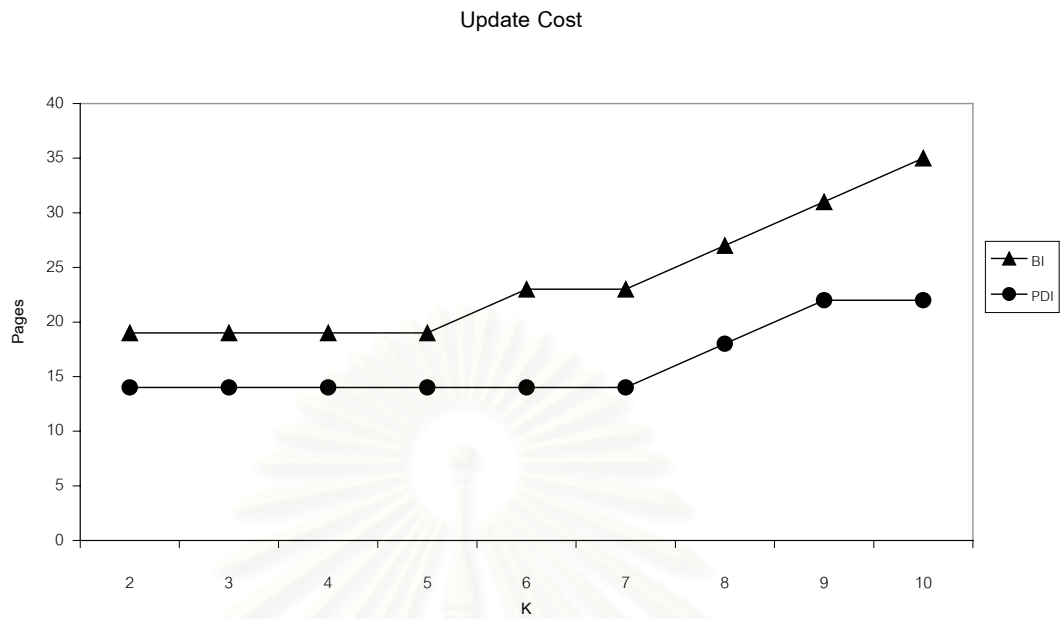


Figure 6.17 The update cost of reference between the *Person* class and the *Vehicle* class

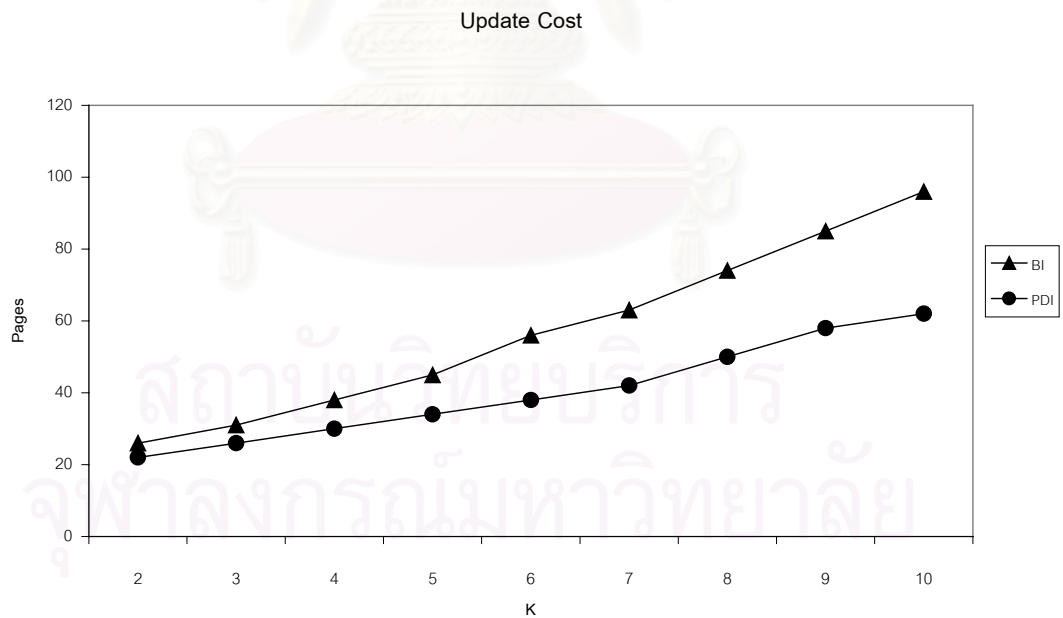


Figure 6.18 The update cost of reference between the *Vehicle* class and the *Company* class

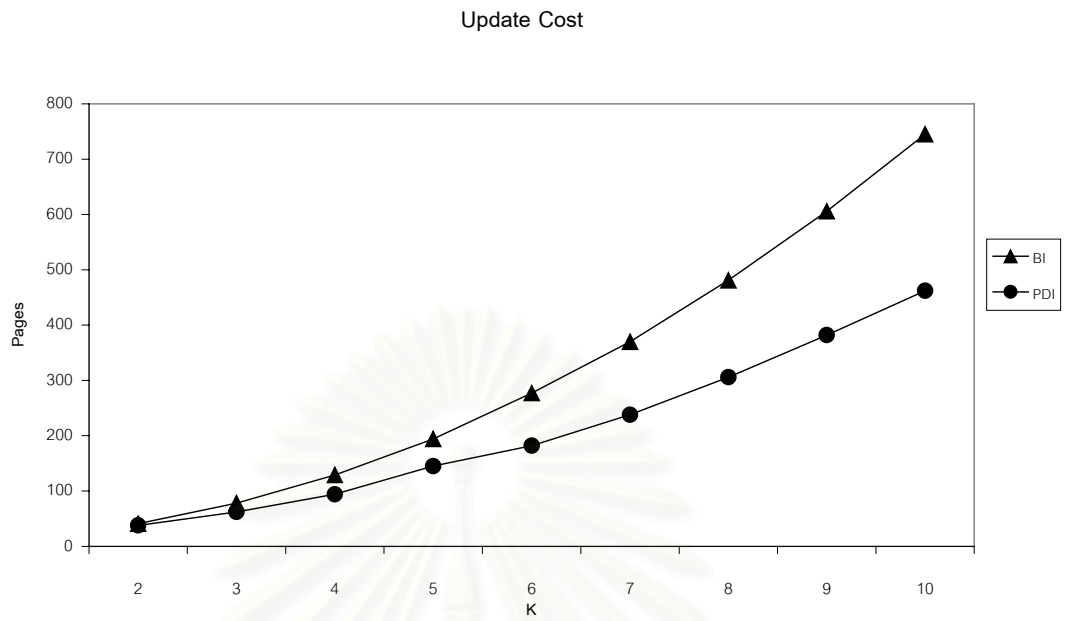


Figure 6.19 The update cost of reference between the *Company* class and the *Bank* class

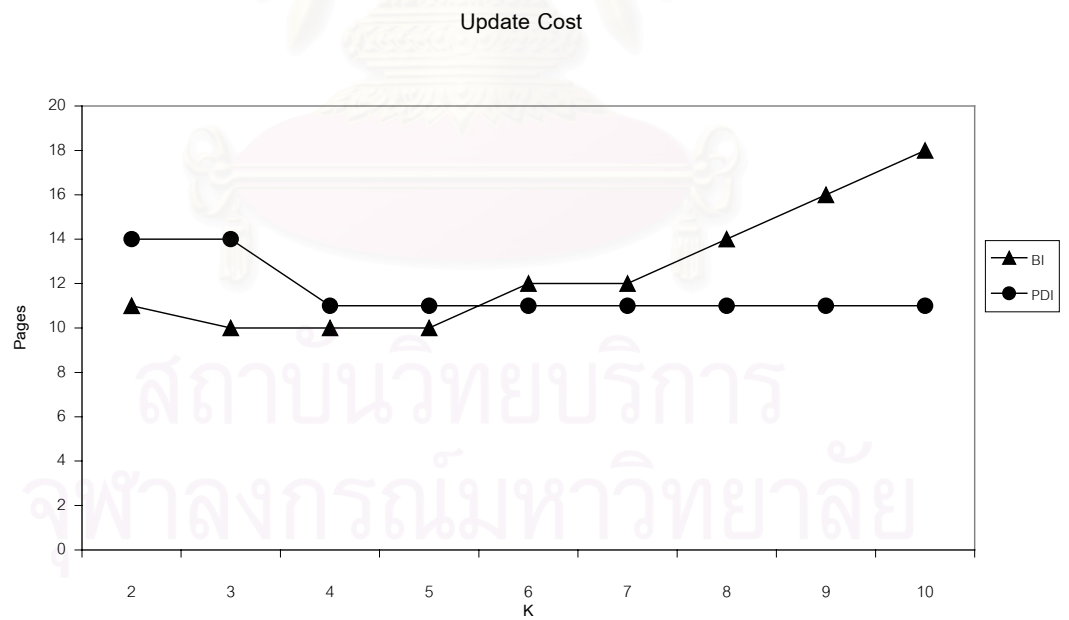


Figure 6.20 The update cost of reference between the *Vehicle* class and the *Engine* class

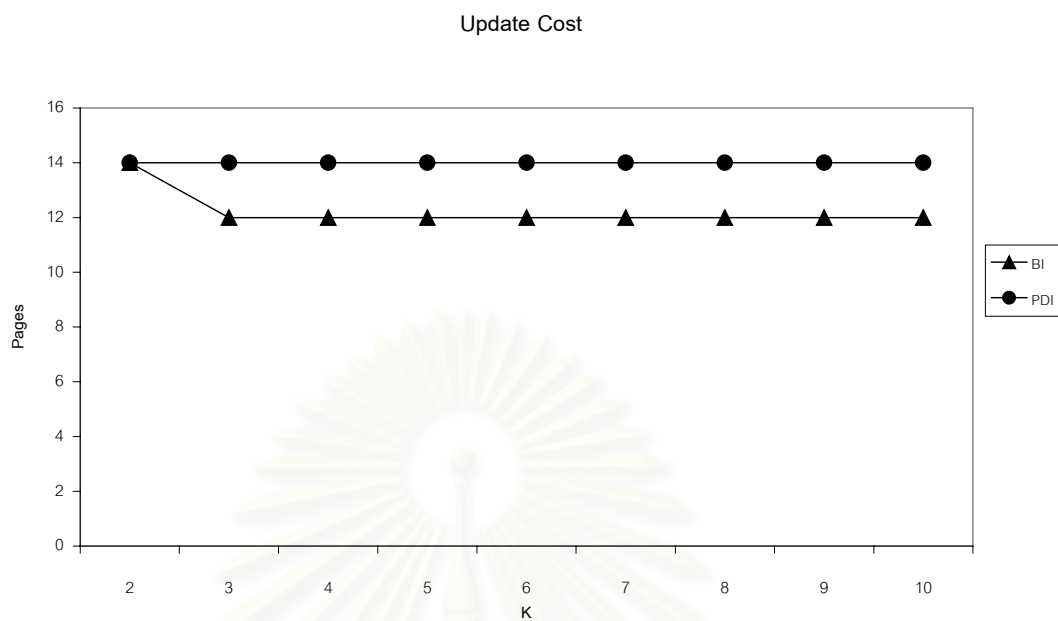


Figure 6.21 The update cost of reference between the *Person* class and the *Course* class

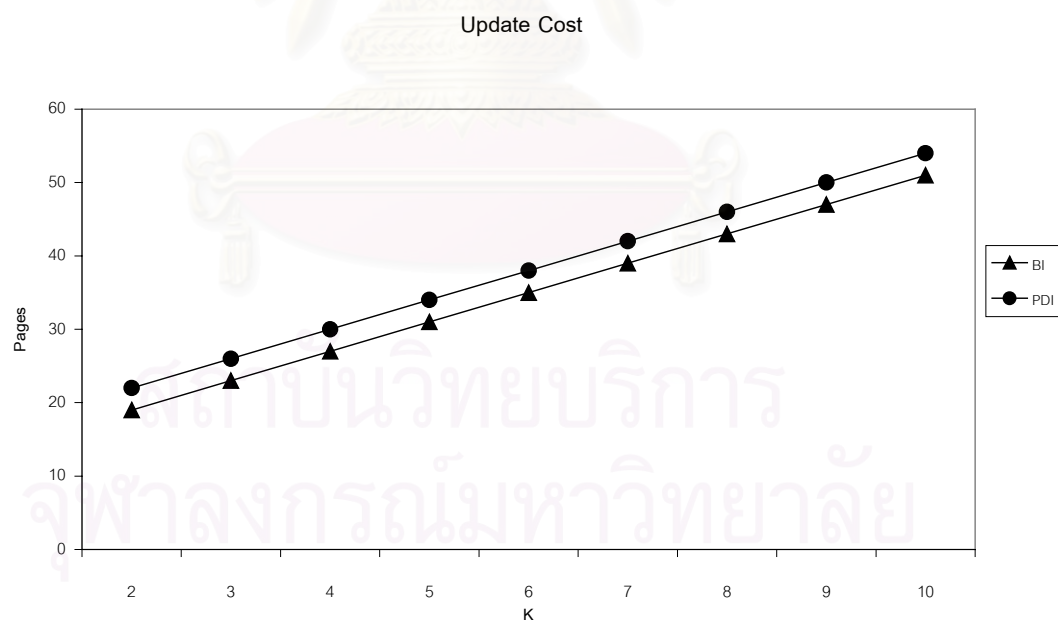


Figure 6.22 The update cost of reference between the *Course* class and the *University* class

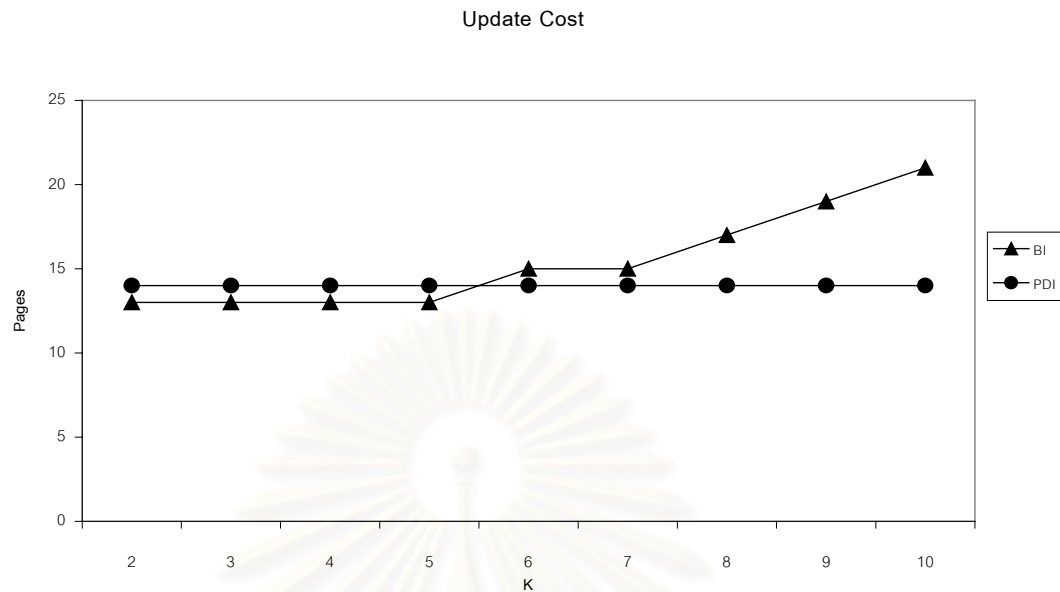


Figure 6.23 The update cost of reference between the *Person* class and the *Computer* class

- Comparison:

It is noticeable from Figure 6.17, Figure 6.18 and Figure 6.19 that the update cost of the Path Dictionary Index (PDI) is better than that of the Branch Index (BI). Since the information stored in the s-expression of the Path Dictionary Index is lower than those stored in the branch information of the Branch Index, update cost of the Branch Index will be higher when we have to read and write back the branch information. As the update is performed between the leaf branch as in Figure 6.20 and Figure 6.23, the update cost of the Branch Index is lower than that of the Path Dictionary Index when the value of K is less than 6. That is because when K is less than 6, the storage of the branch information is not more than a page and only the identity index of the leaf branch need to be updated. It is no need to update the ancestor objects. However, as the value of K is more than 5, the benefit gained is not sufficient when compared with the higher storage of the branch information. When update operation is perform on the child branch as in Figure 6.21 and Figure 6.22, The update cost of the Branch Index is lower than that of the Path Dictionary Index. That is because the level of non-leaf node of the identity index of the *University* class and the *Course* class of the

Branch Index is smaller than that of the identity index of the path 3 of the Path Dictionary Index.

The next chapter will conclude the research and give the perspective for the further research.



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

CHAPTER 7

CONCLUSION AND PERSPECTIVE

This chapter concludes the research for the access methods of the aggregation hierarchy as a tree and proposes the possible research in this area for the future.

7.1 Conclusion

The aggregation hierarchy as a tree is considered as a more complicated form than a path. Therefore, the efficient access method to handle the query on the aggregation hierarchy should be developed. This research introduced the new access methods called the Direct Access to Terminal Virtual Path (DTVP), the Virtual Path Signature (VPS) and the Branch Index (BI) to evaluate the query on an aggregation hierarchy as a tree and then compared them with the Path Dictionary Index method for the path scheme. The reference sharing of classes and the shared key values were varied to observe the storage cost, the retrieval cost and the update cost of these access methods.

Path dictionaries were created to mimic the branches generated from the algorithm of the branch generation for the comparison. The attribute indexes were created on one of a simple attribute of the root class and one of a simple attribute of the leaf class of a sample of the aggregation hierarchy. The identity indexes were also created for the objects of every class on a branch of the Branch Index and on a path of the Path Dictionary Index for the complex attribute searching. The result of the storage cost is that the storage cost of the Direct Access to Terminal Virtual Path (DTVP) is the lowest and the cost of the Virtual Path Signature (VPS) is the second lowest. The storage cost of the Branch Index (BI) is much lower than that of the Path Dictionary Index (PDI) because the information of the leaf branch can be stored as part of the complete branch so that the redundant path is eliminated.

It can be concluded from the comparison of Chapter 6 that when the predicate is specified on the index attribute of the root class and the target is on any leaf classes, the Direct Access to Terminal Virtual Path is the most appropriate than any other access methods. However, if the predicate is specified on other classes, the Direct Access to Terminal Virtual Path is inapplicable. Although the retrieval cost of the Virtual Path Signature is high when compared with other methods, the Virtual Path Signature is the most flexible if we do not know if the predicate is specified on any attributes of any classes. Comparing the retrieval cost of the Branch Index and the Path Dictionary Index, the cost of the Branch Index is apparently lower than that of the Path Dictionary Index when the target class is on the ancestor path or branch of the predicate class. The retrieval cost of the Branch Index is slightly higher than that of the Path Dictionary Index when the predicate class and the target class are on the same path dictionary or branch and the entry size of the branch information is bigger than a page size. Generally speaking, if we do know that the predicate and the target class are on the same path, the Path Dictionary Index is more appropriate. However, if the predicate can be specified on any classes of the aggregation hierarchy as a tree and the entry size of the branch information is not more than a page size, the Branch Index is the most suitable of all access methods mentioned earlier.

As explained in Chapter 6, the entry size of a branch may be bigger than the s-expression of a compared path dictionary because more information, such as the leaf branch and the associated parent branch, is stored in a branch. Therefore, the update cost of the branch index is higher than the update cost of the path dictionary index when the update of reference is performed on that branch. However, if the entry size of the branch information is not more than a page size, the update of reference between classes on different branches of the Branch Index is lower than that on a path of the Path Dictionary Index.

It can be concluded that the Branch Index is more appropriate for the aggregation hierarchy as a tree than the Path Dictionary Index, especially when the retrieval operation is high and especially when the entry size of the branch information involved is not bigger than a page size. The Branch Index can be reduced to the form of the Path Dictionary Index when there is only one path of the aggregation hierarchy and

then all cost will be the same. Therefore, the Branch Index is more general than the Path Dictionary Index.

7.2 Perspective

Throughout the research, there are some assumptions that limit the general case for the access method. Actually the value of an attribute can be multi value or a multiple sets of values. A complex query, such as those predicates with the logical expression ‘AND’ or ‘OR’ are also a challenge ones for query processing on the aggregation hierarchy. Furthermore, the access method for the most complicated form of the aggregation hierarchy as a graph should be in consideration in the future.



REFERENCES

1. Banerjee, J; Kim, W.; and Kim, K.-C. Queries in Object-Oriented Databases. Proceedings of 4th International Conference on Data Engineering, pp. 31-38, 1988.
2. Bartels, D. ODMG 93- The Emerging Object Database Standard. Proceedings of 12th International Conference on Data Engineering, pp. 674-676, 1996.
3. Bertino, E., and Foscoli, P. Index Organizations for Object-Oriented Database System. IEEE Transaction on Knowledge and Data Engineering, Vol. 7, No. 2, 1995: 193-209.
4. Bertino, E., and Foscoli, P. On Modeling Cost Functions for Object-Oriented Databases. IEEE Transaction on Knowledge and Data Engineering, Vol. 9, No. 3, 1997: 500-508.
5. Bertino, E., and Guglielmina, C. Optimization of Object-Oriented Queries Using Path Indices. Proceedings of 2nd International Workshop Research Issues on Data Engineering, pp. 140-149, 1992.
6. Bertino, E., and Kim, W. Indexing Technique for Queries on Nested Objects. IEEE Transaction on Knowledge and Data Engineering, Vol. 1, No. 2, 1989: 196-214.
7. Bertino, E.; Negri, M.; Pelagatti, G.; and Sbattella, L. Object-Oriented Query Languages: The Notion and the Issues. IEEE Transaction on Knowledge and Data Engineering, Vol. 4, No. 3, 1992: 223-237.
8. Chen, Y.-H., and Chang, A.J.T. Object Signatures for Supporting Efficient Navigation in Object-Oriented Databases. Proceedings of 8th International Workshop on Database and Expert System Application, pp. 502-507, 1997.
9. Cho, W.-S.; Lee, S.-S.; and Yoon, Y.-I. A Join Algorithm Utilizing Multiple Path Indexes in Object-Oriented Database Systems. Proceedings of 2nd International Conference on Engineering of Complex Systems, pp. 376-382, 1996.
10. Choenni, S.; Bertino, E.; Blahken, H.M.; and Chang, T. On the Selection of Optimal Index Configuration in OO Databases. Proceedings of 10th International Conference on Data Engineering, pp. 526-537, 1994.

11. Deux, O. et al. The Story of O₂. IEEE Transaction on Knowledge and Data Engineering, Vol. 2, No. 1, 1990: 91-108.
12. Fotouhi, F.; Lee, T.-G.; and Grosky, W.I. The Generalized Index Model for Object-Oriented Database Systems. Proceedings of 10th Phoenix Conference on Computer and Communication, pp. 302-308, 1991.
13. Gude, E. A Uniform Indexing Scheme for Object-Oriented Databases. Proceedings of 12th International Conference on Data Engineering, pp. 238-246, 1996.
14. Han, J.; Xie, Z.; and Fu, Y. Join Index Hierarchy: An Indexing Structure for Efficient Navigation in Object-Oriented Databases. IEEE Transaction on Knowledge and Data Engineering, Vol. 11, No. 2, 1999: 321-337.
15. Hua, K.A., and Tripathy, C. Object Skeleton: An Efficient navigation Structure for Object-Oriented Database System. Proceedings of 10th International Conference on Data Engineering, pp. 508-517, 1994.
16. Ioannidis, Y.E. Query Optimization. ACM Computing Surveys, Vol. 28, No. 1, 1996: 121-123.
17. Ishikawa, Y., and Kitagawa, H. Analysis of Indexing Schemes to Support Set Retrieval of Nested Objects. Proceedings of International Symposium on Advanced Database Technologies and Their Integration, 1994.
18. Kilger, C, and Moerkotte, G. Indexing Multiple Sets. Proceedings of 20th International Conference on VLDB, pp. 180-191, 1994.
19. Kim, W. Object-Oriented Databases: Definition and Research Directions. IEEE Transaction on Knowledge and Data Engineering, Vol. 2, No.3, 1990 : 327-341.
20. Kim, K.-C.; Kim, W.; and Dale, A. Cyclic Query Processing in Object-Oriented Databases. Proceedings of 5th International Conference on Data Engineering, pp. 564-571, 1989.
21. Lee, D.L., and Lee, W.-C. Using path Information for a Query Processing in Object-Oriented Database Systems. Proceedings of Conference on Information and Knowledge Management, pp. 64-71, 1994.
22. Lee, D.L., and Lee, W.-C. Signature Path Dictionary for Nested Object Query Processing. Proceedings of 15th International Conference on Computers and Communications, pp. 275-281, 1996.

23. Lee, W.-C., and Lee, D.L. Signature File Methods for Indexing Object-Oriented Database Systems. Proceedings of 2nd International Computer Science Conference, pp. 616-622, 1992.
24. Lee, W.-C., and Lee, D.L. Short Cuts for Traversals in Object-Oriented Database Systems. Proceedings of Internaitonal Computer Symposium, pp. 1172-1177, 1994.
25. Lee, W.-C., and Lee, D.L. Combining indexing Technique with Path Dictionary for Nested Object Queries. Proceedings of 4th International Conference on Database Systems for Advanced Applications, pp. 107-114, 1995.
26. Lee, W.-C., and Lee, D.L. Path Dictionary: A New Access Method for Query Processing in Object-Oriented Databases. IEEE Transaction on Knowledge and Data Engineering, Vol. 10, No.3, 1998: 371-388.
27. Low, C.C.; Ooi, B.C.; and Lu, H. H-trees: A Dynamic Associative Search Index for OODB. Proceedings of SIGMOD International Conference on Management of Data, pp. 134-143, 1992.
28. Mahatthanapiwat, P., and Rivepiboon, W. Direct Access to Terminal Virtual Path in OODB. Proceedings of National Computer Science and Engineering, 1999.
29. Mahatthanapiwat, P., and Rivepiboon, W. Virtual Path Signature: An Approach for Flexible Searching in OODB. Proceedings of International Conference on Intelligent Technology, pp. 335-340, 2000.
30. Mahatthanapiwat, P., and Rivepiboon, W. Branch Index: An Approach for Query Processing in OODB. International Journal of Information Technology, Vol. 7, No. 2, 2001.
31. Maier, D.; Stein, J.; Otis, A.; and Purdy, A. Development of an Object-Oriented DBMS. OOPSLA '86 Proceedings, pp. 472-482, 1986.
32. Seo, S.K., and Lee, Y.J. Optimal Configuration of Nested Attribute Indexes in Object-Oriented Databases. Proceedings of 20th EUROMICRO Conference on System Architecture and Integration, pp. 379-386, 1994.
33. Shidlovsky, B., and Bertino, E. A Graph-Theoretic to Indexing in Object-Oriented Databases. Proceedings of 12th International Conference on Data Engineering, pp. 230-237, 1996.

34. Shin, H., and Chang, J. A New Signature Scheme for Query Processing in Object-Oriented Database. Proceedings of 20th International Conference on Computer Software and Applications, pp. 400-405, 1996.
35. Sreenath, B., and Seshadri, S. The hcC-tree: An Efficient Index Structure for Object-Oriented Databases. Proceedings of 20th International Conference on VLDB, pp. 203-213, 1994.
36. Sung, S.Y., and Fu, J. Access Methods on Aggregation of Object-Oriented Database. Proceedings of International Conference on Systems, Man and Cybernetics, Vol. 2, pp. 977-982, 1996.
37. Taniar, D. Forward vs. Reverse Traversal in Path Expression Query Processing. Proceedings of Technology of Object-Oriented Languages, pp. 127-140, 1998.
38. Xie, Z., and Han, J. Join Index Hierarchy for Supporting Efficient Navigation in Object-Oriented Databases. Proceedings of 20th International Conference on VLDB, pp. 522-533, 1994.
39. Young, H.-S.; Lee, S.; and Kim, H.-J. Applying Signatures for Forward Traversal Query Processing in Object-Oriented Databases. Proceedings of 10th International Conference on Data Engineering, pp. 518-525, 1994.



APPENDICES

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

APPENDIX I

COST MODEL OF THE PATH DICTIONARY INDEX

Parameters :

$N_{i,j}$: The number of objects in class j of path dictionary i .

$A_{i,j}$: The complex attribute of class j on path dictionary i .

$D_{i,j}$: Distinct value of complex attribute $A_{i,j}$.

$UIDL$: The length of Object Identifier.

P : Page size.

pp : The size of page pointer.

f : Average fan out from a non-leaf node.

kl : Average length of a key value in attribute index.

$OFFL$: The length of offset field in the path dictionary.

SL : The length of start field in the path dictionary.

FSL : The length of free space in path dictionary.

EL : The length of EOS in path dictionary.

$SA_{i,j,k}$: Simple attribute k of class j on path dictionary i .

$U_{i,j,k}$: The number of distinct values for simple attribute $SA_{i,j,k}$.

$q_{i,j,k}$: The ratio of shared attribute value = $N_{i,j}/U_{i,j,k}$.

$k_{i,j}$: Reference sharing of class j on path dictionary i .

Storage Overhead

For the path dictionary i , the average number of objects in an s-expression is:

$$NOBJ = 1 + \sum_{l=1}^{n-1} \prod_{j=l}^{n-1} k_{i,j}.$$

when there are n classes in the path dictionary i .

The average size of an s-expression is:

$$SS = SL * (n - 1) + (UIDL + OFFL) * NOBJ + EL.$$

The number of pages needed for all of the s-expressions on the path is:

$$SSP = \begin{cases} \lceil N_{i,n} / \lfloor P / SS \rfloor \rceil & \text{if } SS \leq P, \\ N_{i,n} * \lceil SS / P \rceil & \text{if } SS > P. \end{cases}$$

The number of pages needed for the free space directory is:

$$FSD = \lceil SSP * (pp + FSL) / P \rceil.$$

The total number of objects in this path dictionary is:

$$TOBJ = NOBJ * N_{i,n}.$$

identity index:

The number of leaf pages needed for path dictionary i is:

$$LP_{iden} = \lceil TOBJ / \lfloor P / (UIDL + pp) \rfloor \rceil.$$

The number of non-leaf pages is:

$$NLP_{iden} = \lceil LP_{iden} / f \rceil + \lceil \lceil LP_{iden} / f \rceil / f \rceil + \dots + x.$$

where $x < f$. If $x \neq 1$, NLP_{iden} is increased by 1 for the root node.

Therefore, the total number of identity index is:

$$IIP = LP_{iden} + NLP_{iden}.$$

attribute index:

The average number of pages needed for a leaf node record is:

$$XP_{SA_i, j, k} = kl + q_{i, j, k} * (UIDL + pp).$$

The number of leaf node pages is:

$$LP_{SA_i, j, k} = \begin{cases} \lceil U_{i, j, k} / \lfloor P / XP_{SA_i, j, k} \rfloor \rceil & \text{if } XP_{SA_i, j, k} \leq P. \\ U_{i, j, k} * \lceil XP_{SA_i, j, k} / P \rceil & \text{if } XP_{SA_i, j, k} > P. \end{cases}$$

The number of non-leaf pages is:

$$NLP_{SA_i, j, k} = \lceil LO_{SA_i, j, k} / f \rceil + \lceil \lceil LO_{SA_i, j, k} / f \rceil / f \rceil + \dots + x.$$

Where $LO_{SA_i, j, k} = \min(U_{i, j, k}, LP_{SA_i, j, k})$ and $x < f$. If $x \neq 1$, $NLP_{SA_i, j, k}$ is increased by 1 for the root node. Therefore, the total number of pages for attribute index is:

$$AIP_{SA_i, j, k} = LP_{SA_i, j, k} + NLP_{SA_i, j, k}.$$

In case of m attribute indexes:

$$AIP = AIP_{index1} + AIP_{index2} + \dots + AIP_{indexm}.$$

Therefore, the storage cost for path dictionary i is:

$$SC_{PDI} = FSD_i + SSP_i + IIP_i + AIP_i.$$

Retrieval Cost

It is assumed that there is only one predicate attribute in the queries and the predicate is specified on the indexed attribute. The retrieval cost of path dictionary index consists of the following:

- Cost of attribute index scanning.
- Cost of accessing the target objects for the qualified s-expressions when the target class is in the same s-expression as the predicate class, otherwise access the qualified join objects to traverse to the target objects in the other path dictionary.

Case 1: the predicate class and the target class are on the same path dictionary

$$RC_{PDI} = h_{attr} + \lceil XP_{attr} / P \rceil + N_{P/Q} * \lceil SS / P \rceil.$$

when h_{attr} is the height of the attribute index -1; for the predicate class. XP_{attr} is the leaf node record of the attribute index. $N_{P/Q}$ is the number of the qualified s-expressions for the predicate P of query Q . SS is an s-expression of the path dictionary.

Case 2: the predicate class and the target class are on different path dictionaries

We can formulate the retrieval cost according to the location of the target class and join class.

- *The target class is an ancestor class of the join class*

$$RC_{PDI} = h_{attr} + \lceil XP_{attr} / P \rceil + N_{P/Q} * \lceil SS_p / P \rceil + N_j * [(h_{iden} + 1) + (SS_t / P)].$$

when SS_p is an s-expression of path dictionary for the predicate class, SS_t is an s-expression of path dictionary for the target class. N_j is the number of qualified join objects of the join class. h_{iden} is the height of the identity index -1; of the join class.

- *The target class is a descendant class of the join class*

We can use the same formula above. Furthermore, we can use the forward traversal from the objects of the join class to the target objects of the target class. However, if the distance between the join class and the target class is high, we should use the identity index of the join class to retrieve the qualified s-expressions to access the target objects

Update Cost

When a complex attribute of one object is updated, Two different cases are categorized as follows.

A. *The class of the updated object or its ancestor classes have no attribute index*

In this case, the update will be performed to the reference between objects. We can use the identity index of the old and new child objects to retrieve the qualified s-expressions and then update the information in the s-expressions. Finally, update of the identity index for the updated object and its ancestor objects have to be performed. We assume that the updated object is on the m^{th} class of the path dictionary.

$$UC_{PDI} = 2 * (h_{iden} + 1 + 2 * \lceil SS / P \rceil) + \left(\sum_{l=1}^{m-1} \prod_{j=l}^{m-1} k_{i,j} + 1 \right) * (h_{iden} + 2).$$

when h_{iden} is the height of the identity index - 1.

B. *The class of the updated object or its ancestor classes have an attribute index*

In addition to all terms in previously cost model, cost for update attribute index should be considered.

$$UC_{PDI} = 2 * (h_{iden} + 1 + 2 * \lceil SS / P \rceil) + \left(\sum_{l=1}^{m-1} \prod_{j=l}^{m-1} k_{i,j} + 1 \right) * (h_{iden} + 2) + (h_{attr} + 2 * \lceil XP_{attr} / P \rceil).$$

when h_{iden} is the height of the identity index-1 and

h_{attr} is the height of the attribute index-1.

APPENDIX II
LETTER OF ACCEPTANCE FROM IJIT



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

APPENDIX II
LETTER OF ACCEPTANCE FROM IJIT

Chief Editor
Robert K.L. Gay

31 October 2001

Prof Pichayotai Mahatthanapiwat
14/250 Moo 4 Ramindra 17
Anusowaree Bangkhen
Bangkok Thailand 10220

Dear Prof Pichayotai

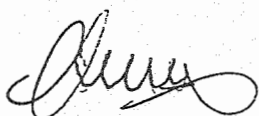
**PAPER TITLED "BRANCH INDEX: AN APPROACH FOR QUERY PROCESSING
IN OODB"**

Please be informed that your paper has been reviewed and accepted. It will be included in the next volume of our online journal.

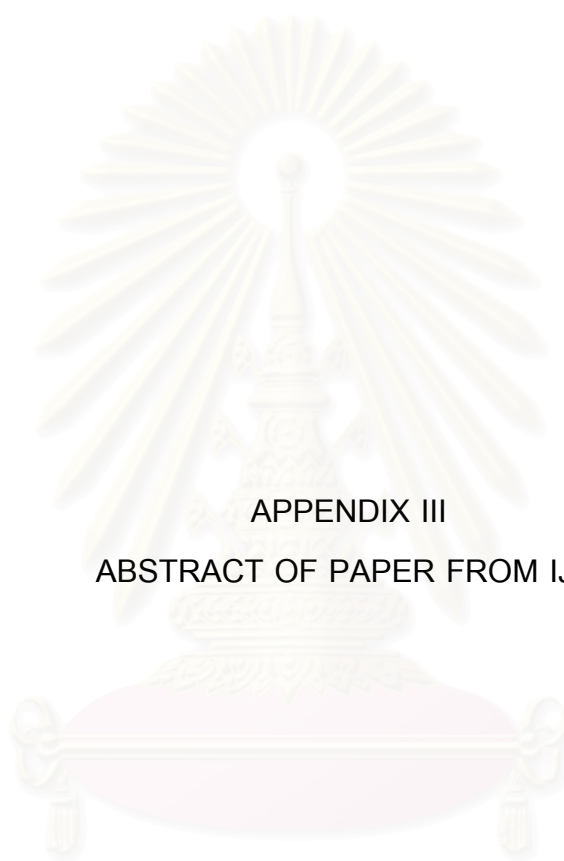
Thank you for submitting your paper for International Journal of Information Technology (IJIT). We look forward to your continued support as we strive for greater excellence.

With best regards

Yours sincerely



Jane Chan
for Prof Robert Gay
Chief Editor, IJIT



APPENDIX III

ABSTRACT OF PAPER FROM IJIT

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Branch Index: An approach for Query Processing in OODB

Pichayotai Mahatthanapiwat and Wanchai Rivepiboon

Department of Computer Engineering

Chulalongkorn University

Bangkok Thailand 10330

p41pmh@hotmail.com, wanchai.r@chula.ac.th

Abstract

In this paper, we present an access method called branch index for query processing of the aggregation hierarchy as a tree in object-oriented databases. The algorithm of branch generation will be proposed to generate all branches for the tree aggregation of classes in the database. For each branch, the information of linking objects is stored so that class traversal methods can be eliminated. Using a set of attribute indexes and identity indexes for each branch, associative searching can be conveniently performed. We discuss the retrieval and update operation and then develop cost models in terms of storage overhead, retrieval cost and update cost. When compared with the path dictionary index for multiple paths, the result shows that our approach has less storage overhead and the retrieval cost is improving.

Key words: access method, object-oriented database, aggregation hierarchy, query processing.

1. Introduction

At present, object-oriented databases have been widely used in most engineering applications, such as Computer Aided Design (CAD), Computer Aided Manufacturing (CAM) and Geographical Information System (GIS). The complexity of data in these applications makes the conventional database, such as the relational database cumbersome to manage them. One of the benefits of the object-oriented database is from its data model [11]. In the object data model, the value of an attribute does not limit to a primitive value, such as integer, real or string, but the value of an attribute can be either a primitive value or a complex value. The complex value of an attribute is a unique Object Identifier (OID) of an object in a class. If a class C consists of an attribute A whose domain is a class C' , the class C can reference the class C' from the attribute A . We call this relation of classes as an aggregation hierarchy. In the same way, the class C' consists of an attribute A' whose domain is a class C'' so that the class C' can link to the class C'' directly and the class C can link to the class C'' indirectly. If a class N is referenced by a class C either directly or indirectly and the class N does not reference any classes, the class N will be called a leaf class of the aggregation hierarchy. On the other hand, a class C will be called the root class of the aggregation hierarchy if it references other classes but it is not referenced by any classes. Any classes in the aggregation hierarchy that are between the root class and the leaf class will be called intermediate classes.

Class traversal methods for an aggregation hierarchy can be performed as forward traversal and reverse traversal. In the forward traversal approach, we start from one class and traverse to its child class by using the value of the complex attribute. On the other hand, the reverse traversal approach traverses up to the parent classes. Usually, the forward traversal approach can perform conveniently because of the inherent pointer of the complex attributes. However, the reverse traversal approach has more trouble unless reverse pointers are implemented between classes. When there is a query, the class that the predicate is involved is called the predicate class and the class of the target objects is called the target class.

If the predicate class and the target class are far away, i.e. there are several intermediate classes between the target class and the predicate class, cost of traversal will be high because of intermediate classes traversal. Therefore, much research has been performed to reduce cost of class traversal whereas the associative searching is also in consideration. The indexing techniques are considered to accelerate database operations by constructing efficient access structures on a database given a certain physical implementation of the database. Secondary index on an attribute or a combination of attributes is useful for evaluating queries on a nested class in an object-oriented database. A classic research on index [1] has been done on an aggregation hierarchy, for example, multi index, nested index, path index. Other [3], [13], [14], [15], [16] researches on the aggregation hierarchy attempted to improve the performance of searching by using the concept form [1]. Indexing techniques on both aggregation hierarchy and inheritance hierarchy are proposed by [4], [8], [9] and [12].

Most indexing techniques that are used for the aggregation hierarchy are proposed as a path scheme. However, for the application that a class schema is more complicated than a path, such as a tree, a new access method should be considered to cope with all classes in the aggregation hierarchy. An example of the aggregation hierarchy that forms a tree of linking classes is shown in Figure 1. It consists of eight classes, *Person*, *Vehicle*, *Company*, *Bank*, *Engine*, *Course*, *University* and *Computer*.

BIOGRAPHY

Mr. Pichayotai Mahatthanapiwat was born on November, 10 1964 and got a Bachelor Degree in Mechanical Engineering (Hon.) at King Mongkut Institute of Technology, Thonburi, Bangkok in 1987. He got a Master Degree in Computer Science at Chulalongkorn University, Bangkok in 1991. At present, he works as a lecturer at School of Computer Engineering, Institute of Engineering, Suranaree University of Technology, Nakhonratchasima.



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย