INDEXING TECHNIQUES FOR OBJECT-ORIENTED DATABASES

By

Frank Hing-wah Luk

A Thesis

submitted in partial fulfillment of the requirements

for the Degree of Master of Philosophy

Division of Computer Science and Engineering

The Chinese University of Hong Hong

1996

香港中文大學
大學
圖書館系統
10 MAY 1997
UNIVERSITY
LIBRARY SYSTEM
THE CHINESE UNIVERSITY OF HONG KONG

Abstract

Object-oriented database management systems have generated significant excitement in the database community. They provide data management facilities for new application areas such as computer-aided design (CAD), computer-aided manufacturing (CAM), knowledge-based systems, AI systems, multimedia applications with images and sound.

However, the support for large and complex data in object-oriented database management systems might downgrade the system performance, which is considered as an important factor on their acceptance in the new application domains.

This thesis addresses the issues of efficient indexing support for query processing in object-oriented database management systems and presents some indexing techniques that directly support the object-oriented paradigm.

An indexing structure "Triple Node Hierarchy" is proposed for optimizing query processing in object-oriented database systems. The structure provides efficient support for object references along an aggregation hierarchy by maintaining direct mapping between the objects of the two classes, while the intermediate objects are maintained separately for update purpose. With suitable modifications, the Triple Node Hierarchy method can provide fast support for object navigation in both aggregation and inheritance hierarchies. Our results have showed that the Triple Node Hierarchy performs better than other methods in object-oriented query processing.

Acknowledgement

First of all, I would like to thank my supervisor Dr. Ada Fu for her guidance, help and encouragement while I was conducting the thesis research. This thesis comes from the numerous inspiring discussions with her.

I thank Dr. Chin Lu, Dr. Lei-zhen Cai and Dr. Siu-cheung Chau for being my examiners. Their comments and suggestions help to improve the thesis greatly.

Thanks go to my fellow students Wing-shun Kan and Edward Tam for going to movies and playing Warcraft with me so many times!

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

Contents

Abstract **ii**

Acknowledgement **iii**

1 Introduction **1**

1.1 Motivation 1

1.2 The Problem in Object-Oriented Database Indexing 2

1.3 Contributions 3

1.4 Thesis Organization 4

2 Object-oriented Data Model **5**

2.1 Object-oriented Data Model 5

2.2 Object and Object Identifiers 6

2.3 Complex Attributes and Methods 6

2.4 Class 8

2.4.1 Inheritance Hierarchy 8

2.4.2 Aggregation Hierarchy 8

2.5 Sample Object-Oriented Database Schema 9

3 Indexing in Object-Oriented Databases **10**

3.1 Introduction 10

3.2 Indexing on Inheritance Hierarchy 10

3.3 Indexing on Aggregation Hierarchy 13
3.4 Indexing on Integrated Support 16
3.5 Indexing on Method Invocation 18
3.6 Indexing on Overlapping Path Expressions 19

4 Triple Node Hierarchy **23**

4.1 Introduction 23
4.2 Triple Node 25
4.3 Triple Node Hierarchy 26
4.3.1 Construction of the Triple Node Hierarchy 26
4.3.2 Updates in the Triple Node Hierarchy 31
4.4 Cost Model 33
4.4.1 Storage 33
4.4.2 Query Cost 35
4.4.3 Update Cost 35
4.5 Evaluation 37
4.6 Summary 42

5 Triple Node Hierarchy in Both Aggregation and Inheritance Hierarchies **43**

5.1 Introduction 43
5.2 Preliminaries 44
5.3 Class-Hierarchy Tree 45
5.4 The Nested CH-tree 47
5.4.1 Construction 47
5.4.2 Retrieval 48
5.4.3 Update 48
5.5 Cost Model 49
5.5.1 Assumptions 51

5.5.2 Storage 52
5.5.3 Query Cost 52
5.5.4 Update Cost 53
5.6 Evaluation 55
5.6.1 Storage Cost 55
5.6.2 Query Cost 57
5.6.3 Update Cost 62
5.7 Summary 63

6 Decomposition of Path Expressions **65**
6.1 Introduction 65
6.2 Configuration on Path Expressions 67
6.2.1 Single Path Expression 67
6.2.2 Overlapping Path Expressions 68
6.3 New Algorithm 70
6.3.1 Example 72
6.4 Evaluation 75
6.5 Summary 76

7 Conclusion and Future Research **77**
7.1 Conclusion 77
7.2 Future Research 78

A Evaluation of some Parameters in Chapter 5 **79**

B Cost Model for Nested-Inherited Index **82**
B.1 Storage 82
B.2 Query Cost 84
B.3 Update 84

C Algorithm constructing a minimum auxiliary set of JIs **87**

D Estimation on the number of possible combinations **89**

Bibliography **92**

List of Tables

4.1 Database Parameters 33

4.2 Application-specific Parameters 37

5.1 System Parameters 49

5.2 Database Parameters 50

5.3 Derived Parameters 51

B.1 Database Parameters 83

List of Figures

2.1 Sample Object-Oriented Database Schema 9

4.1 Triple Node Hierarchy and decomposition graph for P(1,7) . . . 27

4.2 Overlapping Path Expressions 29

4.3 Triple Node Hierarchy supporting Tri(0,4,$\top$) and Tri($\perp$,0,4) . . 29

4.4 Triple Node Hierarchy supporting Tri(0,6,$\top$) and Tri($\perp$,0,6) . . 29

4.5 Triple Node Hierarchy supporting Tri(1',6',$\top$) and Tri($\perp$,1',6') . 30

4.6 Combined Triple Node Hierarchy 30

4.7 Algorithm Update 31

4.8 Algorithm Update_Propagation 32

4.9 Storage Vs small fanout 38

4.10 Storage Vs large fanout 39

4.11 Navigation Cost Vs small fanout 39

4.12 Navigation Cost Vs objects selected 40

4.13 Update Cost Vs small fanout 41

4.14 Navigation Update Mix (fanout set to 1.5) 41

5.1 A leaf node record in CH-tree 46

5.2 nCH-tree organizations 47

5.3 Storage Cost 56

5.4 Retrieval Cost Vs number of classes in the inheritance hierarchies 58

5.5 Retrieval Cost Vs position of class having large fanout 58

5.6 Retrieval Cost Vs position of class having large fanout 59
5.7 Retrieval Cost Vs range of key values covered by the queries . . 60
5.8 Retrieval Cost Vs range of key values covered by the queries . . 60
5.9 Average Retrieval Cost Vs path length 61
5.10 Update Cost 63

6.1 Performance measurement of original algorithm 66
6.2 Decomposition Graph for P(1,7) 67
6.3 Decomposition Graph for P(1,7) 68
6.4 Decomposition Graph for {P(1,7), P(3,6)} 69
6.5 Decomposition Graph for {P(1,7), P(3,9)} 70
6.6 Construction of Decomposition Graph for a set of Overlapping Path Expressions 71
6.7 Overlapping Path Expressions 72
6.8 Decomposition Graph of P(0,4) and P(0,6) 74
6.9 Decomposition Graph of P(1',6') 74
6.10 CPU Time and System Time Vs path length 75

Chapter 1

Introduction

1.1 Motivation

With the advance of the technology of computer systems, a number of technical application areas, such as computer-aided design (CAD), computer-aided manufacturing (CAM), knowledge-based systems, AI systems, multimedia applications with images and sound, benefit from the support of database technologies. These application domains require semantically rich modeling capabilities for representing and manipulating the structurally complex inter-relationships among data [21]. Traditional database systems, such as relational database systems do not adequately support these new application domains. This results in the development of new database technology with more suitable functionality.

The object-oriented database technology supports a rich collection of sophisticated data modeling and manipulation concepts. Intensive research towards the object-oriented database technology extends relational database system concepts with data and procedural abstractions, and embellishes object-oriented programming languages with database capabilities. A large number of object-oriented databases are available commercially or are being developed by many

industry or academic research facilities, e.g. GemStone [10], O2 [1], Objectivity [27], ObjectStore [23], Ontos [28], VERSANT [32]. Such intensive research and commercialization activities demonstrate that object-oriented database systems constitute a promising approach towards supporting the new application domains.

Although the object-oriented database technology addresses the needs of the new applications, the support for large and complex data might downgrade the performance of corresponding systems which is considered as an important factor on the acceptance of the object-oriented technology in the new application domains. Efficient query optimization and access planning had been proven to be a cornerstone of relational systems performance, and will gain even more importance for semantically richer queries and complex data. Thus, object-oriented database application domains must provide excellent performance to meet the challenge due to large volume of complex data, and we believe that efficient indexing is critical in making object-oriented database systems competitive in terms of performance with traditional database systems.

1.2 The Problem in Object-Oriented Database Indexing

Indexing techniques have been widely investigated in the framework of conventional databases, such as relational databases and several organizations have been proposed in the literature. For example, B-trees and hashing are some of the most common ways to implement an index. However, the novel features of object-oriented data models require index organizations beyond conventional techniques in order to provide efficient support for the queries that are possible in databases based on these advanced data models. We therefore need to develop new indexing techniques to support query processing in object-oriented

database systems efficiently.

1.3 Contributions

This thesis focuses on indexing techniques for object-oriented databases and presents new indexing structures to improve the system performance. The main contributions of this thesis include the following research results:

Support for aggregation hierarchy: We proposed an indexing technique, called the Triple Node Hierarchy, which supports fast navigations among objects and classes along path expressions in object-oriented databases.

Overlapping of path expressions in an aggregation hierarchy: In an object-oriented database, more than one index might be constructed on different path expressions along an aggregation hierarchy. We address the problem of overlapping subpaths in path expressions in an aggregation hierarchy, and illustrate how the Triple Node Hierarchy solves the problem.

Support for the integration of aggregation hierarchy and inheritance hierarchy: We integrate the Triple Node Hierarchy with the CH-tree to construct an indexing structure, called the nested CH-tree, which supports efficient retrieval of instances against a class hierarchy on a nested attribute. The nested CH-tree constructs a CH-tree indexed on the nested attribute while the intermediate objects are maintained separately by the Triple Node Hierarchy. We demonstrate that the Triple Node Hierarchy can be integrated with other indexing structures that originally support the inheritance hierarchy. We show that this integration can achieve good performance.

1.4 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 presents an overview of the basic concepts of an object-oriented data model. A comprehensive review of indexing techniques for object-oriented databases is presented in chapter 3. In chapter 4, a new index structure, called the Triple Node Hierarchy, is proposed, which provides efficient support for object navigation along aggregation hierarchy. We have developed a cost model and the results have shown that the proposed structure outperforms other index structures. In chapter 5, we further extend the Triple Node Hierarchy method to provide an integrated support for queries which have access scope along inheritance hierarchy and aggregation hierarchy. In chapter 6, we discuss the details on the decomposition of path expressions. Finally, chapter 7 concludes the thesis with a discussion on future research.

Chapter 2

Object-oriented Data Model

2.1 Object-oriented Data Model

An object-oriented data model is based on five fundamental concepts [8]

- Each real world entity is modeled by an object. Each object is associated with a unique identifier (OID).

- Each object has a set of attributes (or instance variables) and methods (operations), and the value of an attribute can be an object or a set of objects. The set of attributes of an object represents the object structure and the set of methods represents an object behavior.

- The attribute values represent the state of an object. This state is accessed or modified by sending messages to the object to invoke the corresponding methods.

- Objects sharing the same structure and behavior are grouped in classes. A class represents a template for a set of similar objects. Each object is an instance of some class. A class consists of a number of attributes, and the value of an attribute of an object belonging to the class is an object or a set

of objects belonging to an arbitrary class. This results in an aggregation hierarchy. An attribute of any class on an aggregation hierarchy is a nested attribute of the root class in the hierarchy.

- A class can be defined as a specialization of one or more classes. A class defined as specialization is called a subclass. A subclass inherits attributes, messages, and methods from its superclass(es). The superclass/subclass relationship results in an inheritance hierarchy, which is orthogonal to the aggregation hierarchy.

2.2 Object and Object Identifiers

In object-oriented databases, each real world entity is represented by an object. An object is associated a state and a behavior. The state is represented by the values of the attributes of the object while the behavior is defined by the methods acting on the state of the object upon invocation of corresponding operations.

The identity of an object is independent of the values of the object attributes. A system that is identity-based allows an object to be referenced via a unique internally generated number, an object identifier, independent of the attribute values of the object.

For performance reasons, if the domain of an attribute is a primitive class, such as integers or characters, the values of the attribute are directly represented; that is, instances of a primitive class have no identifiers associated with them.

2.3 Complex Attributes and Methods

In object-oriented databases, the domain of an attribute can be any class: both primitive and non-primitive. Object attributes may have complex values, such as sets or reference to other object. There are three kinds of complex attributes:

reference attributes, collection attributes, and derived attributes.

Reference attributes, or associations, are used to represent relationships between objects. They take on values that are objects. Normally, the object identifiers of the object entities that are being referenced are stored as the attribute values. Reference attributes are analogous to pointers in a programming language, or to foreign keys in a relational system [11]. The ability to take an object as the value for an attribute greatly simplifies the modeling of a database and makes the modeling more straightforward and natural.

Collection attributes is used for lists, sets, or arrays of values. The collections may include simple attribute values and also references. Operations are provided for creating, inserting, or deleting an element from a collection. Many object-oriented database systems support collections which reflect a real world need, for instance, to describe a person's hobbies(set), ordered preferences(list), etc.

Derived attributes are those whose values can be defined procedurally rather than stored explicitly, by specifying a procedure to be executed when the value is retrieved or assigned. For example, we may store such personal information as birth date and age in a personnel database. The birth date will not change but the age does. It would be desirable to define a procedure for the age attribute so that it always represents the difference between the current date and the birth date.

Methods are procedures used in object-oriented databases to encapsulate or "hide" the attributes of an object, providing the only interface to manipulate the object. This encapsulation provides a form of 'logical data independence' and means that the implementation of objects can be modified, without affecting the applications that use them.

2.4 Class

Class is used to group together objects that respond to the same message, use the same methods, and have variables of the same name and type. Each such object is called an instance of its class. All objects in a class share a common definition, though they differ in the value assigned to the variables [22].

Briefly, a class defines the structure (the attributes and relationships in which objects having this type can participate) and behavior (the methods associated with the type) of objects of a particular type.

2.4.1 Inheritance Hierarchy

In an object-oriented database scheme, it is often the case that several classes are similar. It would be desirable to define a representation for the common variables of these classes in one place. To do so, we place classes in a specialization hierarchy, in which a class, called subclass, is defined as a specialization of other class, called superclass. A subclass inherits all the attributes and methods of its superclass and can define its own attributes and methods. If a class inherits attributes and methods from only one class, this inheritance is called single inheritance. Otherwise, it is called multiple inheritance. In a system which supports single inheritance, the classes form a hierarchy, called inheritance hierarchy.

2.4.2 Aggregation Hierarchy

There exists another kind of hierarchy relating to classes, the 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: a class consists of a set of attributes; the domains of some or all of the attributes may be classes with their own sets of attributes, and so on. This definition of a class results in a directed

graph of classes rooted at that class, the aggregation hierarchy. However, this is not a hierarchy in the strict sense of the world, since the classes can be defined recursively.

2.5 Sample Object-Oriented Database Schema

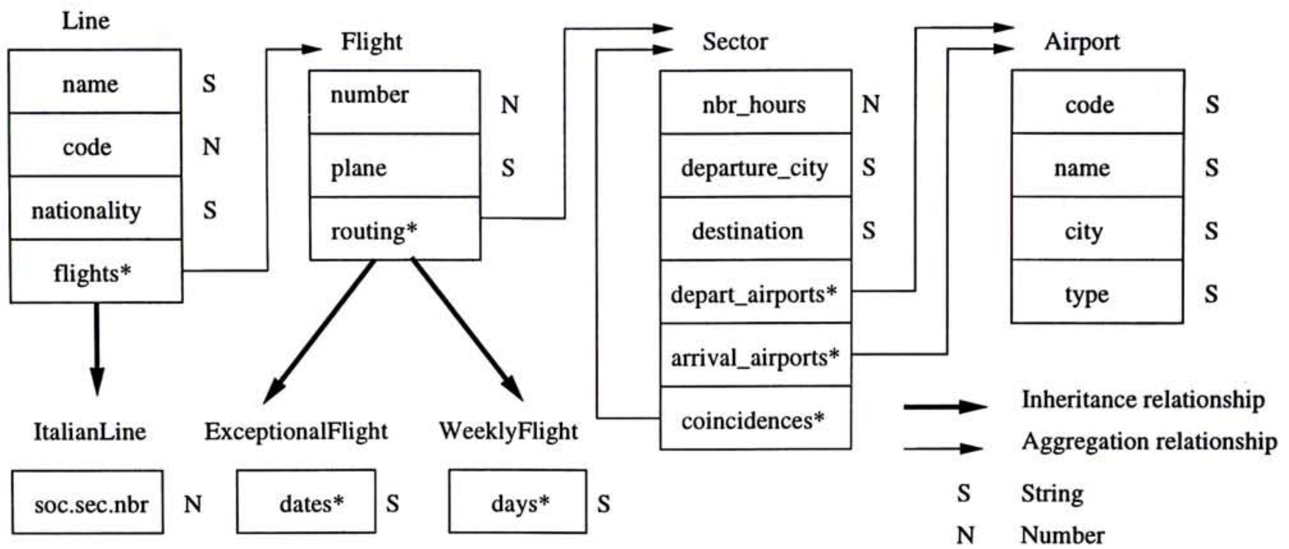


Figure 2.1: Sample Object-Oriented Database Schema

An example of object-oriented database schema is shown in figure2.1. In figure2.1, a class is represented by a box, and the attributes of the class are inside the box. Those attributes labeled with * denote multi-valued attributes. Two types of arcs are used in the representation. A simple arc from a class C to a class C' denotes that C' is the domain of an attribute of C. A bold arc from a class C to a class C' indicates that C' is a subclass of C.

Chapter 3

Indexing in Object-Oriented Databases

3.1 Introduction

Indexing techniques in object-oriented databases can be classified as structural and behavioral. Structural indexing is based on object structure and behavioral indexing is based on object behavior. Structural indexing techniques support queries issued against aggregation hierarchy and inheritance hierarchy while behavioral indexing techniques support queries containing method invocations.

3.2 Indexing on Inheritance Hierarchy

In an inheritance hierarchy, an instance of a subclass is also an instance of its superclass. Hence, the access scope of a query against a class may generally include not only its instances, but also those of its subclasses. Two examples of such queries against the database schema of Figure 2.1 are shown in the following:

- *Retrieve all Italian lines whose nationality is British.*

- *Retrieve all lines (including the Italian ones) whose nationality is British.*

The first query targets on only the class *ItalianLine* while the second query targets on all classes rooted at the class *Line*.

Queries based on some attributes of a class or a class hierarchy can also be classified as *point queries* and *range queries*. Point queries basically ask for all instances of a class or a class hierarchy with a particular value for the concerned attribute. Range queries, on the other hand, ask for instances of a class or a class hierarchy whose attribute value falls in a certain range.

Kim et al. proposed the Class-Hierarchy tree (CH-tree) [20] which is based on B^+-trees and essentially maintains only one index tree for all classes of a class hierarchy. It clusters the OIDs of objects of all classes in the class hierarchy by the value of an indexed attribute. It has simple searching and update algorithms which are similar to those of B^+-trees. The performance for point query is excellent, although range query may be inefficient. Also, searching for values in a single class is treated in the same way as searching for values in a hierarchy of classes.

Low et al. presented an indexing structure called the H-tree [25] which is similar to B^+-trees. A H-tree is maintained for each class of the hierarchy. To index a class hierarchy on a common attribute, the H-tree of the root class of the hierarchy is nested with the H-trees of all immediate sub-classes of the root, and the H-trees of the subclasses are nested with the H-trees of their respective subclasses and so on. H-trees cluster the OIDs of objects of a single class with a given value of the indexed attribute together.

To search on a single class for instances which satisfy the search condition, a H-tree is searched like a B^+-tree by ignoring the nested tree pointers. For searching on multiple classes or the entire class hierarchy, the search begins on the H-tree of the root class, and follows the pointers to search the nested subtrees of classes of interest.

The disadvantage of the H-tree is that the nesting of tree structure is rather complex and contains many physical pointers which make the tree difficult to implement.

Kilger and Moerkotte introduced a set grouping index structure called the CG-tree [19] which maintains one tree and groups objects by the key values. Also, objects of the same class in a hierarchy are clustered together. The leaf pages of a CG-tree contain doubly-linked lists which link up all the indexes of single classes. The second level pages contain a special directory which is maintained on indexed sets. The upper levels are the same as the B_+-tree. The performance of the CG-tree on range searching is improved.

Sreenath and Seshadri also presented a similar index structure called the hierarchy-class Chain tree (hcC-tree) [30]. The hcC-tree clusters the OIDs of objects of a single class and the class hierarchy with a given value of the indexed attribute together by storing information in two kinds of chains - hierarchy chain and class chain. To index a class hierarchy on a common attribute, one hcC-tree is constructed.

To search on a single class for instances which satisfy the search condition, the hcC-tree searches the class chains. To search on a class hierarchy, the hcC-tree searches the hierarchy chain. The drawback of the hcC-tree is that all data stored in class chains are replicated in the hierarchy chain, which makes the size of the hcC-tree extremely large.

Ramaswamy and Kanellakis proposed a class-division index [29] which is an extension of the CH-tree. It maintains unions of extends of the classes in the class hierarchy. Experimental results show that indexing by class-division performs better than the CH-tree for range searching, giving a small space tradeoff.

3.3 Indexing on Aggregation Hierarchy

In an aggregation hierarchy, the value of an attribute of an object is an object or a set of objects. An attribute of any class on an aggregation hierarchy is logically an (nested) attribute of the root of the hierarchy. Hence, the access scope of a query may include the nested attributes, which are usually described in terms of path expressions. A path expression is basically a linear object reference chain leading from one object instance to another. It describes a branch in an aggregation hierarchy.

Definition. Given an aggregation hierarchy H, a path P is defined as

$$P = C_1.A_1.A_2...A_n \ (n \leq 1)$$

where:

- C_1 is a class in H;
- A_1 is an attribute of a class C_1;
- A_i is an attribute of a class C_i in H, such that C_i is the domain of attribute A_{i-1} of class C_{i-1}, $1 < i \leq n$;

Two examples of such queries against the database schema of Figure 2.1 are presented as follows:

- *Retrieve all Italian lines whose nationality is British.*
- *Retrieve all Italian lines having flights Boeing747.*

The first query is issued on the non-nested attribute *nationality* of the class *ItalianLine* while the second query is issued on the nested-attribute *plane.*

Queries based on some attributes of a class may also be classified as *forward queries* and *backward queries.* Forward queries have a search predicate which is based on the attributes (either nested or non-nested). Backward queries, on the

other hand, have a search predicate which is based on the target class. Here are two examples of such queries against the database schema of Figure 2.1:

- *Retrieve all Italian lines whose nationality is British.*
- *Retrieve all planes of Italian lines whose name is British Airways.*

The first query is a backward query having a search predicate based on the *nationality* attribute while the second query is a forward query based on an instance of *ItalianLine.*

Maier and Stein [26] proposed an index organization for which a series of index components, indexes on each level of the nested attributes, are maintained for update propagations. Bertino and Kim [7] presented three index structures: the nested index, path index and multiindex. Nested index provides a direct association between an ending object and the corresponding starting objects along a path, and can be implemented using some variation of the B-tree structure or some hashing algorithms. A path index can be used to evaluate nested predicates on all classes along a path. In both the nested index and the path index, the key values can be instances of a primitive class or a non-primitive class, depending on the domain of the last attribute in the path. A path may be split into several subpaths, and a different index may be used for each subpath. A multiindex is an index which is built on top of a set of nested indexes. The problem of using the above three indexing techniques is that the cost will be very high when the path length is greater than three, and thus a long path is required to be divided into subpaths of a shorter length.

Choenni et al. [13] proposed an optimal index configuration by splitting a long path expression into some shorter ones, and by indexing the shorter paths with the index structures in [7, 2].

Chawathe, Chen and Yu [12] took index interaction into consideration when selecting a set of nested indexes for nested object hierarchies. Index interaction

refers to the phenomenon that the inclusion of one index might have impact on the benefit obtained by the other indexes if the former is overlapped with the latter ones. The problem of selecting an optimal index scheme is formulated as an optimization problem against an objective function. Experiments showed that the index selection improved the overall performance. These approaches only support associative retrieval of objects through nested attributes but not navigations in both directions along a reference chain.

Kemper and Moerkotte [16, 17] presented a data structure, called the access support relation, which keeps the identifiers of the objects connected by the attribute relationship in a path expression and can span over the reference chains of a path expression. The access support relation is very similar to conventional indexes except that it provides entry points from both directions. Several alternatives including full, canonical, left and right extensions and the decomposition of access support relations for a given path expression are discussed. The optimal one is determined according to some domain-specific information such as probabilities of different types of queries and updates. The storage size of each component in an access support relation could be large because all the identifier sequences of the joinable objects along an object path corresponding to the component are stored, and because any two objects in two classes could be connected by more than one object path. Further, an update on one object may need to be propagated to several components or to the entire access support relation, which could be costly.

Hua and Tripathy [14] proposed a navigation structure, call the object skeleton, which essentially is a network of object identifiers. Two object identifiers are connected if the corresponding objects are associated by, for example, an attribute relationship. This approach is more general in the sense that the navigations can be supported between two classes not only in a path expression but also over a network of classes. The navigations, however, are supported

efficiently only if the starting points of the navigations can be located by using some nested indexes such as those in [7, 2].Besides, an update is required to be propagated over the network of object identifiers and the nested indexes.

Xie and Han proposed the Join Index Hierarchy method [33]which constructs a hierarchy of join indexes. Each join index node stores pairs of object identifiers of two classes that are connected via direct or indirect logical relationships. Three types of join index hierarchies including complete, partial and based, are discussed. Based join index hierarchy is too simple while complete join index hierarchy is too large. Partial join index hierarchy supports only frequently used navigations, has reasonably small space and update overheads, and speeds up query processing considerably in both forward and backward navigations.

3.4 Indexing on Integrated Support

Although extensive research has been done on object-oriented databases indexing techniques, and various index organizations have been proposed in the literature for supporting queries on aggregation/inheritance hierarchies, as mentioned before, few of them discuss how the organizations are used when both aggregation hierarchy and inheritance hierarchy are taken into account. Queries having access scope of both aggregation and inheritance hierarchies may be any combination of the following four types:

1. forward Vs backward
2. nested attribute Vs non-nested attribute
3. single class Vs class hierarchy
4. point query Vs range query

Two examples of such queries against the database schema of Figure 2.1 are listed in the following:

- *Retrieve all Italian lines having a flight spending less than 2 hours on routing.*
- *Retrieve all lines (including the Italian ones) having a flight departing from the airport named Kai-Tak.*

The first query is a backward range query target on a single class *ItalianLine* issued against the nested attribute of the path expression

P=Line.flights.routing.nbr_hours

The second query is a backward point query target on all classes rooted at the class *Line* issued against the nested attribute name of the path expression

P=Line.flights.routing.departure_city.name

Bertino extended the work on nested index [7] to handle inheritance of classes appearing in a path expression [2]. The index provides an efficient evaluation of queries on nested attributes which target on a single class and on any number of classes in a given inheritance hierarchy. In other words, it provides an integrated treatment of indexing in the framework of both aggregation and inheritance hierarchies.

The structure is similar to a B^+-tree. It groups the object identifiers according to the key value of the tail attribute. Moreover, objects belonging to the head class and the intermediate objects in the path expression are grouped together in a class directory and stored in the primary record.

An auxiliary index, which basically keeps the direct reference information between objects, together with the information in primary records are used to propagate updates.

Although the work was defined only for paths having all single-valued attributes, Bertino and Foscoli removed this restriction and extended the method to support multi-valued attributes, called Nested-Inherited Index structure [5].

3.5 Indexing on Method Invocation

Methods can be used in queries as a derived attribute method or as a predicate method. A derived attribute method has a function similar to that of an attribute, which returns an object or a value upon execution. A predicate method returns a boolean value (True or False). The result can be used to evaluate boolean expression that determines which objects satisfy the query. Processing a query containing a method invocation may cause a method to be executed for each instance of the queried class, and the execution cost may be very difficult to be estimated. Therefore, we need an index technique based on precomputing method results to speed up the query processing.

In Kifer et al. [18], queries containing method invocations are expressed as method expressions, which are similar to path expressions. Traversing along method expressions results in an object or a set of objects (including primitive objects, i.e. values). Several techniques based on the precomputation of methods along method expressions have been proposed. The results are stored in an index or other access structures, so that queries containing method invocations can be efficiently evaluated, because the cost of method execution is transferred to compile time.

In Jhingran [15], a quantitative analysis undertaken in the extended relational system POSTGRES showed that separate caching of precomputed POSTQUEL attributes is almost always superior to caching within the tuples. In Kemper et al., precomputed function results are stored in a separate data structure called Access Support Relation [16, 17] and the results are disassociated from the argument objects. In Bertino [3], precomputed function results are stored in those objects upon which methods are invoked. This technique provides a greater flexibility with respect to object allocation and clustering. The drawback is that this technique can be applied to objects with methods which do not use properties of other objects. In Bertino and Quarati [9], the technique

was extended to remove this restriction.

In general, any indexing techniques, which express the relationship between an object and the results of method expressions, can be useful in processing queries that involve method invocations.

A major issue of this approach is how to detect when the computed method results are no longer valid. In order to do this, in most of the approaches proposed, dependency information is kept. This keeps track of which objects, and possibly, which attributes of objects, have been used to compute a given method. When an object is modified, all the precomputed results of the methods which have used this object are invalidated. Various solutions have been proposed for the problem of dependency information, also in terms of the characteristics of the method.

3.6 Indexing on Overlapping Path Expressions

Several indexing schemes have been proposed to support fast navigation operations between objects along the path expression [7, 16, 17, 33]. With an indexing scheme, a special data structure is constructed to provide direct association between an ending object and the corresponding starting object along the path expression. The intermediate objects on the path expression are also stored, which are mainly used for update propagation.

As there may be more than one indexed path expressions in the aggregation hierarchy, some of the path expressions may be overlapped.

Definition. Given two path expressions P_1 and P_2:

- $P_1 = C_1.A_1.A_2...A_n$
- $P_2 = C'_1.A'_1.A'_2...A'_m$

where C_1 not necessarily equal to C'_1. They are overlapping if there exists

a subscript $j < min(n, m)$ such that the domain of $A_i = A'_j$ $(i \geq 1, j \geq 1)$ and $A_{i+r} = A'_{j+r}$ (r=1,2,3,...).

Two examples of such queries against the database schema of Figure 2.1 are:

- *Retrieve all Italian lines having a flight departing from the airport named Kai-Tak. (Q1)*
- *Retrieve all Italian lines having a flight arriving at the airport named Kai-Tak. (Q2)*

They represent two path expressions

- P_1=Line.flights.routing.depart_airports.name
- P_2=Line.flights.routing.arrival_airports.name

The subpath expression P'=Line.flights.routing is common in P_1 and P_2.

For a general case, consider the two overlapping path expressions:

- $P_a = \underbrace{\underbrace{C_1.A_1...A_i}_{C_i}.A_{i+1}...A_{i+j}}_{C_{i+j}}.A_{i+j+1}...A_n$
- $P_b = \underbrace{\underbrace{C'_1.B_1...B_l}_{C_i}.A_{i+1}...A_{i+j}}_{C_{i+j}}.B'_1...B'_q$

If the domain of the subpath expressions $C_1.A_1...A_i$ and $C'_1.B_1...B_l$ is the same, say C_i , then the two path expressions P_a and P_b are overlapped and they share the overlapping subpath $C_i.A_{i+1}...A_{i+j}$.

It is clear that if we construct two indexes on P_1 and P_2 separately without considering the overlapping condition, the cost of update on the overlapping subpath of the two path expressions may become higher than expected. This leads to great impact on the performance.

Bertino [6] proposed a splitted configuration which splits the path expressions into several subpaths and allocates an index on each subpath. Kemper and Moerkotte [16] proposed a decomposition policy which decomposes two path expressions into five partitions and construct five access support relations on each partition. The five partitions are:

1. $t_0.A_1...A_i$
2. $t_i.A_{i+1}...A_{i+j}$
3. $t_{i+j}.A_{i+j+1}...A_n$
4. $s_0.B_1...B_l$
5. $t_{i+j}.C_1...C_q$

The major drawback of the splitted configuration and decomposition is that the performance on navigation will degrade because additional join operations are needed to derive the full relation.

Xie and Han proposed the Join Index Hierarchy method [33] which decompose a path expression into binary pairs and construct a hierarchy of Join indexes based on these. This method can be extended to consider the overlapping of different path expressions in the aggregation hierarchy.

Clearly, the effect of index interaction needs to be taken into consideration when a set of indexes is built along the aggregation hierarchy. It is noted that as the granularity of data objects in an object-oriented databases becomes finer and the database schema tends to become more sophisticated nowadays, it has become increasingly important to explore the effect of building a set of indexes in the aggregation hierarchy [12].

Note that there may exist more than one semantic linkage between two object classes. For example, queries $Q1$ and $Q2$ (in previous page) represent two kinds of semantic associations between the class *Line* and the class *Airport* which

cannot be mixed up since they carry different semantics (different path expressions). The schema paths should be stored in the schema (data dictionary) with the identification (such as by labeling) of each semantic linkage.

Chapter 4

Triple Node Hierarchy

4.1 Introduction

Query processing and optimization are crucial to the performance of object-oriented database systems. The support of navigation among different classes and objects via class inheritance hierarchies and aggregation hierarchies is also essential. Navigations from one object in a class to objects in other classes, using object identity references, involve a lot of associative search for objects on secondary memory. Such navigations can lead to significant performance degradation.

Different indexing techniques have been proposed to optimize the object reference operation of queries. Some techniques have access scope of instances of all classes in the inheritance hierarchy [20, 2, 24, 25, 30], and some have access scope of nested attributes in classes in the aggregation hierarchy, which can span over the reference chains of a path expression [7, 16, 17, 33].

It is noted that most of the previous work only considered indexing along a single path in an aggregation hierarchy. However, the effects of two indexes could be entangled, that is, the inclusion of one index could affect the benefit achievable by another index. This phenomenon is known as index interaction

[12]. Index interaction has a larger performance impact when the global effect of indexing multiple query paths is considered. When many indexes are evaluated, the interaction among indexes significantly complicates the method to evaluate their costs and benefits globally.

Following the study of Access Support Relation and Join Index Hierarchies, a new indexing structure, called the Triple Node Hierarchy is proposed in this thesis. The Triple Node Hierarchy supports object references by maintaining direct association between two objects along a path expression of arbitrary path length while the intermediate objects are stored separately. It had been shown in [33] that only frequently referenced object pairs should be maintained. Moreover, the Triple Node Hierarchy supports multiple object pairs along multiple query paths of an aggregation hierarchy, minimizing the costs on overlapping path expressions. The Join Index Hierarchy, on the other hand, considers only the simple case of a single path expression.

The Triple Node Hierarchy has the following advantages:

First, in object-oriented database systems, one of the big performance penalties is the object references using object identifiers which may scatter along a reference chain. This object reference operation may involve object faults and disk read operations at widely scattered locations. The Triple Node Hierarchy supports direct mapping between two objects along a path expression, and provides a single lookup mechanism which reduces the I/O cost significantly. Moreover, intermediate objects are stored separately for maintenance purpose.

Second, The Triple Node Hierarchy provides a set-oriented navigation mechanism which is more efficient than traditional object-at-a-time navigation.

Third, The Triple Node Hierarchy supports forward and backward navigations efficiently.

Forth, The Triple Node Hierarchy considers the effect of indexes on overlapping paths. The storage and update costs of constructing additional indexes is

minimized.

4.2 Triple Node

Given a path expression $P = C_1.A_1.A_2...A_n$, a triple node, denoted as Tri(i,j,k) $(1 \leq i < j < k \leq n+1)$ consists of a set of tuples <OID(O_j), OID(O_t), TYPE, m> where O_j is an object which belongs to class C_j and O_t is an object which belongs to either class C_i or class C_k, distinguished by the flag TYPE, and there exist m≥ 1 distinct object paths connecting the two objects O_j and O_t. A B^+-tree is constructed for the triple node, indexed on the OIDs of objects of class C_j.

If there exist tuples $<$ OID(O_j) , OID(O_i), TYPE = i, $m_0 >$ and $<$ OID(O_j) , OID(O_k), TYPE = k, $m_1 >$ in Tri(i,j,k), then there exist $m_0 \times m_1$ distinct object paths connecting O_i and O_k via O_j.

If there exists a tuple $<$ OID(O_j) , OID(O_i), TYPE = i, $m_0 >$, but there do not exist any tuples $<$ OID(O_j) , OID(O_k), TYPE = k, $m_1 >$ in Tri(i,j,k), then there exist object paths connecting O_i and O_j but not emanating to any object in C_k, and vice versa.

The triple node Tri($\perp$,j,k) contains only tuples <OID(O_j), OID(O_k), m> and the triple node Tri(j,k,$\top$) contains only tuples <OID(O_k), OID(O_j), m>. In other words, $\perp$ and $\top$ denote non-exist classes.

It is noted that all attributes in the path except A_n have as domain non-primitive classes. The domain of the last attribute A_n may be either primitive or non-primitive. If it is non-primitive, i.e., it is a value, the B^+-tree that is associated with the triple node Tri(i,n,$\top$), $i \leq n$, will be indexed on the value. In this case, queries may be targeted on instances with a particular value or values that fall in a certain range for attribute A_n.

4.3 Triple Node Hierarchy

The Triple Node Hierarchy is a hierarchy connecting triple nodes together. Normally, the triple node Tri($\perp$,j,k) is used to support forward object references from a set of objects of class C_j to objects of class C_k, while the triple node Tri(j,k,$\top$) is used to support backward object references from a set of objects of class C_k to objects of class C_j. Moreover, update operations are propagated from triple nodes of lower levels.

Suppose the triple node Tri($\perp$,i,k) is constructed to support the forward object references between objects of classes C_i and C_k on the path expression $P(i,k) = C_i.A_i...A_{k-1}$ (or Tri(i,k,$\top$) for backward object references), in order to support update operations of the triple node Tri($\perp$,i,k) (or Tri(i,k,$\top$)), a triple node Tri(i,j,k) , i<j<k, is constructed. Whenever there is an update on Tri(i,j,k), Tri($\perp$,i,k) (or Tri(i,k,$\top$)) is also updated. Two triple nodes Tri(i,p,j) and Tri(j,q,k), where $i < p < j$, and $j < q < k$ are then constructed to support updates on Tri(i,j,k). This relation connects a set of triple nodes together to form a Triple Node Hierarchy.

4.3.1 Construction of the Triple Node Hierarchy

A decomposition graph describes the sequence of decomposing a long path expression into shorter ones. For example, Figure 4.1 shows the decomposition graph of the path expression P(1,7), where

P(1,7) is decomposed at 4 into P(1,4) and P(4,7),

P(1,4) is decomposed at 2 into P(1,2) and P(2,4),

P(2,4) is decomposed at 3 into P(2,3) and P(3,4),

P(4,7) is decomposed at 5 into P(4,5) and P(5,7), and

P(5,7) is decomposed at 6 into P(5,6) and P(6,7).

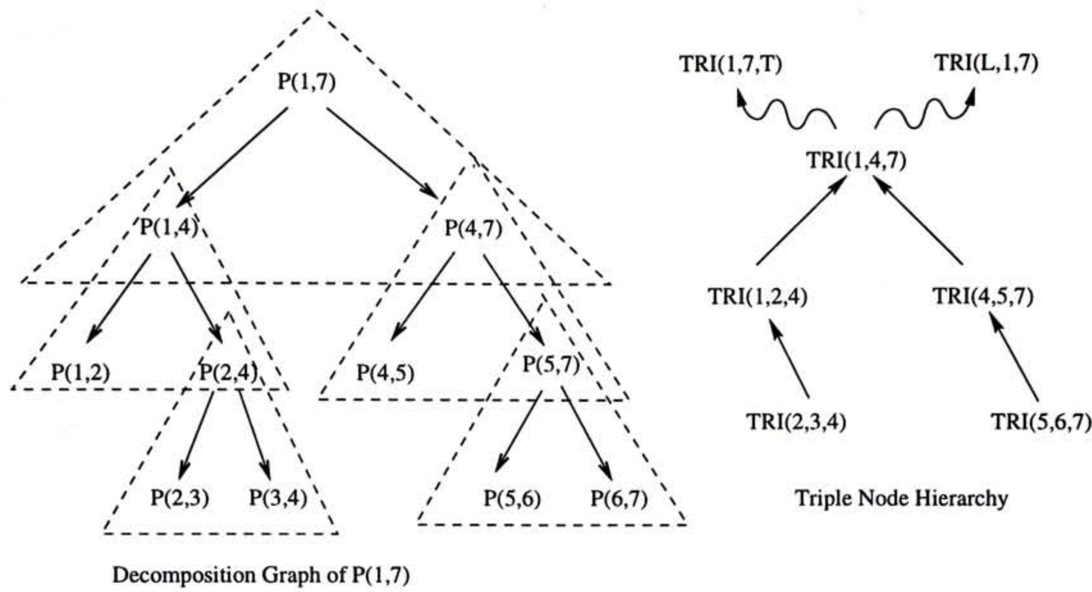


Figure 4.1: Triple Node Hierarchy and decomposition graph for P(1,7)

An optimal decomposition graph is the one which minimizes (1) the total number of subpath expressions in the graph, and (2) the total length of the path expressions in the decomposition graph. It can be generated by the algorithm proposed in [33]. We will discuss the details and improvements on the generation of a decomposition graph in Chapter 6.

Given a decomposition graph, a set of triple nodes can be identified by the decomposition points in the graph. Suppose the path expression P(i,k) is decomposed at j into two subpath expressions, a triple node Tri(i,j,k) is constructed.

For example, in Figure 4.1, P(1,7) is decomposed at 4 into P(1,4) and P(4,7). Thus, the triple node Tri(1,4,7) is included in the Triple Node Hierarchy. P(1,4) is decomposed at 2 into P(1,2) and P(2,4) and thus Tri(1,2,4) is also included. Similarly, the triple nodes Tri(2,3,4), Tri(4,5,7) and Tri(5,6,7) are also included.

After identifying the set of triple nodes in the Triple Node Hierarchy, the level of a triple nodes Tri(i,j,k) is defined as the maximum value of j-i and k-j. For example, Tri(2,3,4) is level 1, Tri(1,2,4) is level 2 and Tri(1,4,7) is level 3.

Therefore, the Triple Node Hierarchy can be constructed easily by constructing the triple nodes from the lowest level up. A level 1 triple node Tri(i,i+1,i+2)

can be computed directly as classes C_i, C_{i+1} and C_{i+2} are connected by direct logical relationship. The result is indexed on class C_{i+1}. Higher levels triple nodes can also be computed when their lower level triple nodes are computed. Finally, the triple node Tri($\perp$,s,t) or Tri(s,t,$\top$) will be constructed.

Thus, the Triple Node Hierarchy can be constructed as shown in Figure4.1. To construct a Triple Node Hierarchy supporting the triple node Tri(1,7,$\top$) (or Tri($\perp$,1,7)) on the path expression P(1,7), Tri(1,4,7) is constructed, which is in turn supported by the two triple nodes Tri(1,2,4) and Tri(4,5,7). Tri(1,2,4) is then supported by Tri(2,3,4) and Tri(4,5,7) is supported by Tri(5,6,7).

Note that some systems do not store information between an object and the objects referencing it. In these systems, we need to store extra triple nodes Tri(i,i+1,$\top$) ($1 \leq i \leq n$), for every class C_{i+1} in the path expression $P = C_1.A_1.A_2...A_n$ except those where there is a triple node Tri(i,i+1,Y) (Y>i+1) being stored in the Triple Node Hierarchy.

The reason why we organize the decomposition graph in the form of a Triple Node Hierarchy instead of constructing join index nodes on each subpath expression to form a Partial Join Index Hierarchy is that update costs can be reduced in the process of update propagation. For example, suppose there is an update in the relationship P(5,6) in a Partial Join Index Hierarchy, the update will propagate upward to P(5,7) by performing a join operation on P(5,6) and P(6,7). This extra cost on join operations is reduced in the Triple Node Hierarchy as the triple node Tri(5,6,7) already stores the results. It can be demonstrated in the following section.

The Triple Node Hierarchy can be used to support a set of frequently referenced object pairs. These object pairs lie on a set of path expressions in the aggregation hierarchy which may be overlapped.

Consider the case shown in Figure 4.2. Suppose we have to support three

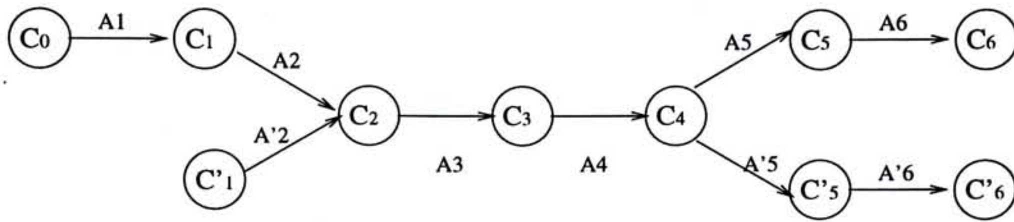


Figure 4.2: Overlapping Path Expressions

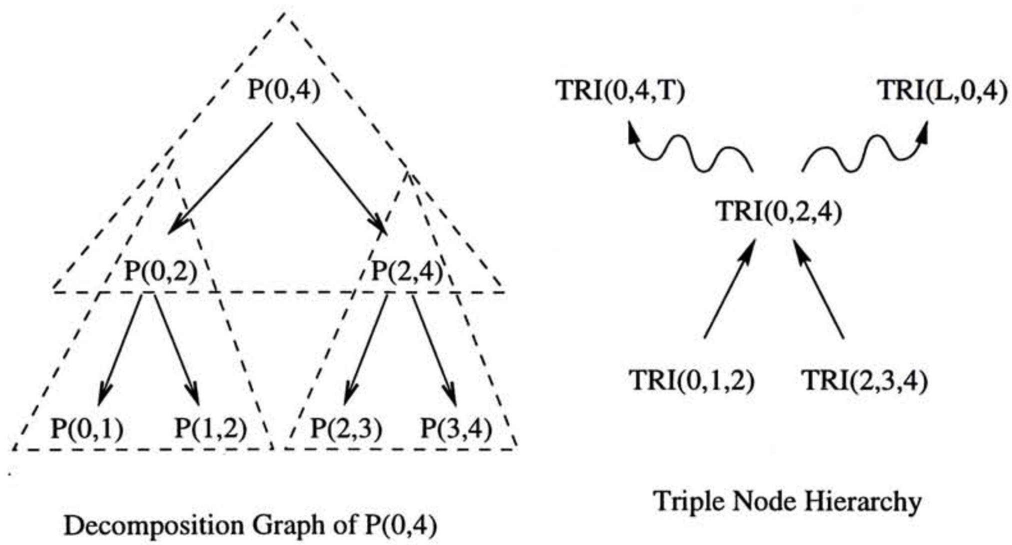


Figure 4.3: Triple Node Hierarchy supporting Tri(0,4,$\top$) and Tri($\perp$,0,4)

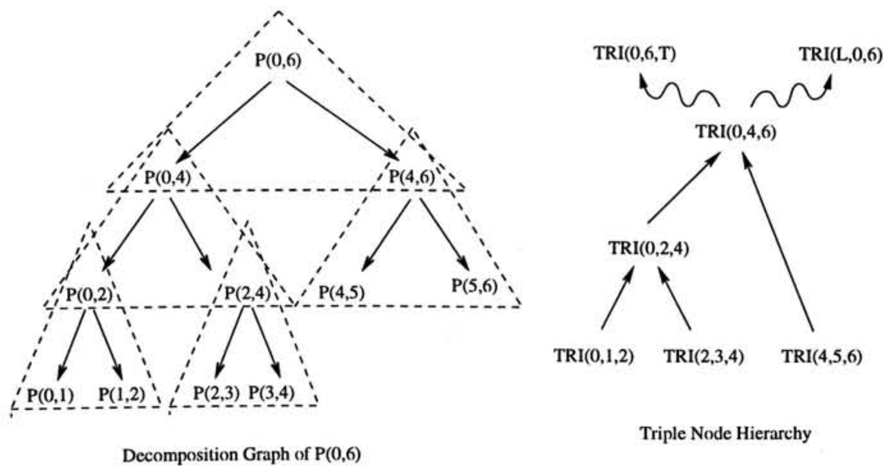


Figure 4.4: Triple Node Hierarchy supporting Tri(0,6,$\top$) and Tri($\perp$,0,6)

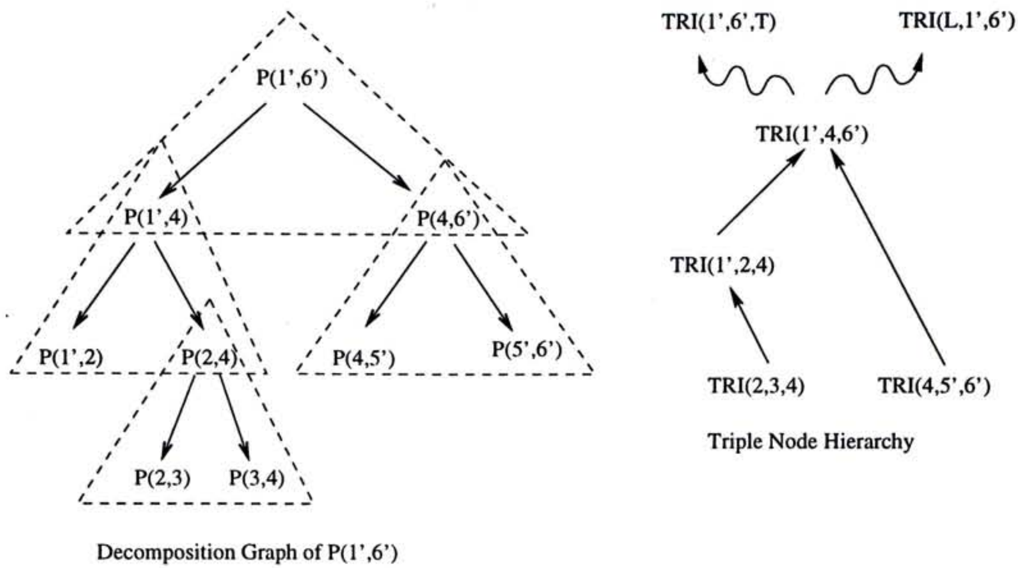


Figure 4.5: Triple Node Hierarchy supporting Tri(1',6',$\top$) and Tri($\perp$,1',6')

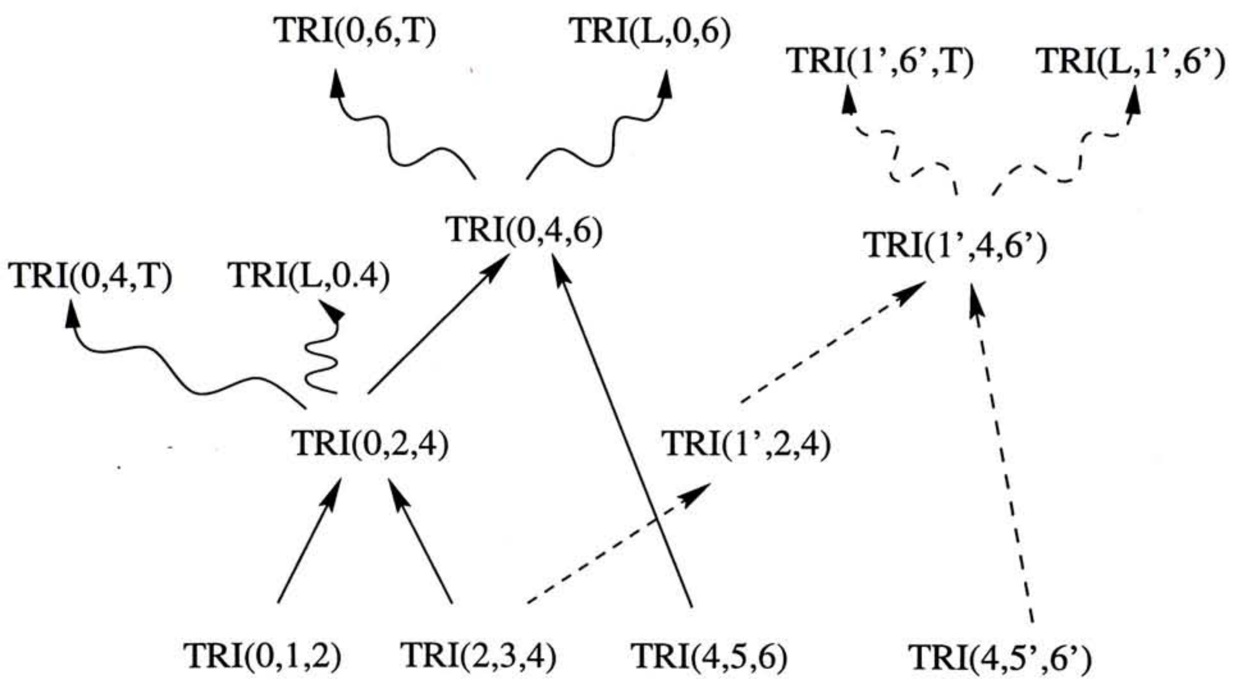


Figure 4.6: Combined Triple Node Hierarchy

frequently referenced object pairs on P(0,4), P(0,6) and P(1'6'), the decomposition graphs of the three path expressions and the corresponding Triple Node Hierarchies are shown in Figure 4.3,Figure 4.4 and Figure 4.5. Figure 4.6 shows the combined Triple Node Hierarchy.

Update(δ(i,j), Tri(x,i,j))

Input: a set of update operations δ(i,j)
a triple node Tri(x,i,j)
Output: a set of update operations δ(x,j)

for all tuples $(OID(O_i), OID(O_j), m)$ in δ(i,j)
  if there exists tuple $< OID(O_i), OID(O_j), TYPE = j, m_1 >$ in Tri(x,i,j)
    set Δm=m_1+m
    if Δm > 0
      replace the tuple with $< OID(O_i), OID(O_j), TYPE = j, \Delta m >$
    else
      remove the tuple
    endif
  else
    set Δm=m
    insert tuple $< OID(O_i), OID(O_j), TYPE = j, \Delta m >$
  endif
  for all tuples $< OID(O_i), OID(O_x), TYPE = x, m_0 >$ in Tri(x,i,j)
    insert $(OID(O_x), OID(O_j), m_0 \times \Delta m)$ into δ(x,j)
  endfor
endfor

Figure 4.7: Algorithm Update

4.3.2 Updates in the Triple Node Hierarchy

Updates in object-oriented databases will be reflected in the index structure. An update in the relationship between two classes will affect the corresponding triple nodes in the Triple Node Hierarchy.

Here we consider three types of update operations in the attribute relationship A_i between two classes C_i and C_j :

Insert operation : insert O_{i+1} into $O_i.A_i$

This operation inserts an object O_{i+1} of class C_{i+1} into the attribute A_i of an object O_i of class C_i.

Delete operation : delete O_{i+1} from $O_i.A_i$

This operation deletes an object O_{i+1} of class C_{i+1} from the attribute A_i of an

---

Update_Propagation(δ(i,j))
Input: a set of update operations δ(i,j)

```
for all triple nodes in the triple node hierarchy
   for all triple nodes with the format Tri(x,i,j),
      Update(δ(i,j), Tri(x,i,j))
      Update_Propagation(δ(x,j))
   endfor
   for all triple nodes with the format Tri(i,j,y),
      Update(δ(i,j), Tri(i,j,y))
      Update_Propagation(δ(i,y))
   endfor
endfor
```

Figure 4.8: Algorithm Update_Propagation

object O_i of class C_i.

Modify operation :

This operation modifies the attribute value of an object O_i of class C_i from an object O_{i+1} of class C_{i+1} to another object O'_{i+1} of class C_{i+1}. This operation is performed by deleting O_{i+1} from $O_i.A_i$ and inserting O'_{i+1} into $O_i.A_i$

δ(i,j) denotes a set of update operations on classes C_i and C_j. δ(i,j) contains a set of tuples $(OID(O_i), OID(O_j), m)$, where m denotes the total number of object paths connecting O_i and O_j which are being updated, and a positive m implies that the operation is an insert operation while a negative m implies delete operation.

Figure 4.7 shows an algorithm which performs a set of update operations δ(i,j) in a triple node Tri(x,i,j) where $x < i < j$. The update operations in triple node Tri(i,j,Y) where $i < j < Y$ can be done similarly.

Figure 4.8 shows an algorithm which performs an update in the Triple Node Hierarchy. Update operations are always initiated by a set of update operations δ(i,i+1), and propagated to higher level triple nodes.

Parameters	semantics, derivation/default
$\|C_i\|$	number of objects in class C_i
f_i	average number of references from an object in C_i to objects in C_{i+1}
$shar_i$	average number of objects in class C_i referencing the same object in C_{i+1}, $= \frac{\|C_i\| \times f_i}{\|C_{i+1}\|}$
P	net size of page, $= 4096$
$OIDL$	size of object identifier, =8
tnc	size of the counter in a tuple of a triple node, =4
$type$	size of the TYPE flag in a tuple of a triple node, =1
tn	size of a tuple in a triple node, $=OIDL \times 2 + type + tnc$
PP	size of page pointer, =4
α	average page occupancy factor, =70%
B_{fan}	fan out of the B^+ tree, $B_{fan} = \lceil \frac{P \times \alpha}{PP+OIDL} \rceil$

Table 4.1: Database Parameters

4.4 Cost Model

An analytical model is constructed to study the performance of the Triple Node Hierarchy. The study focuses on several crucial performance measurements, including storage size, navigation cost and update cost (propagated over the Triple Node Hierarchy). Table 4.1 lists some system parameters used in the cost analysis.

4.4.1 Storage

The probability that an object in class C_{j-1} does not reference a particular object in class C_j is

$$\frac{\begin{pmatrix} |C_j| - 1 \\ f_{j-1} \end{pmatrix}}{\begin{pmatrix} |C_j| \\ f_{j-1} \end{pmatrix}} = 1 - \frac{f_{j-1}}{|C_j|}$$

The probability that m objects in class C_{j-1} do not reference a particular

objects in class C_j is

$$(1 - \frac{f_{j-1}}{|C_j|})^m$$

The probability that a particular object in class C_j is referenced by m objects in class C_{j-1} is

$$1 - (1 - \frac{f_{j-1}}{|C_j|})^m$$

Therefore, the number of objects in class C_j referenced by these m objects in class C_{j-1} is

$$|C_j| \times (1 - (1 - \frac{f_{j-1}}{|C_j|})^m)$$

Hence, the average number of distinct objects in class C_j referenced by a set of m objects in class C_i is

$$fwd(i,j,m) = \begin{cases} \lceil p(|C_j|, f_{j-1}, m) \rceil & , \quad j = i+1 \\ \lceil p(|C_j|, f_{j-1}, fwd(i, j-1, m)) \rceil & , \quad j > i+1 \end{cases}$$

and the average number of distinct objects in class C_i referencing a set of m objects in class C_j is

$$bwd(i,j,m) = \begin{cases} \lceil p(|C_i|, shar_i, m) \rceil & , \quad j = i+1 \\ \lceil p(|C_i|, shar_i, bwd(i+1, j, m)) \rceil & , \quad j > i+1 \end{cases}$$

where $p(x,y,z) = x \times (1 - (1 - \frac{y}{x})^z)$

The number of tuples in a triple node Tri(⊥,i,j) (and Tri(i,j,⊤)) is

$$|Tri(\perp, i, j)| = |C_i| \times fwd(i, j, 1)$$

The number of tuples in a triple node Tri(i,j,k) is

$$|Tri(i,j,k)| = |C_j| \times fwd(j,k,1) + |C_i| \times fwd(i,j,1)$$

The approximate number of pages needed to store a triple node The number of pages needed to store a triple node is

$$\|Tri(i,j,k)\| = \lceil \frac{tn \times |Tri(i,j,k)|}{P \times \alpha} \rceil$$

4.4.2 Query Cost

From Yao [34], for accessing k records randomly distributed in a file of n records stored in m pages, the expected optimal number of page access is given by

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

Following Valduriez [31], the number of disk access for a forward navigation from a set of class n_i objects in class C_i to objects in C_j in a triple node Tri($\perp$,i,j) is

$$1 + Yao(ni, \lceil \frac{|C_i|}{B_{fan}} \rceil, |C_i|) + Yao(ni, \|Tri(\perp, i, j)\|, |C_i|)$$

It is assumed that a B^+-tree is of two levels. In the above equation, the constant "1" indicates that one page is accessed for retrieving the root node. The second term is the number of page access to the leaf pages of the B^+-tree in order to find the page pointers for n_i object identifiers. The last term is the number of page access required to find the corresponding n_i object identifiers. The number of disk access for a forward navigation from a set of n_j objects in class C_j to objects in class C_k in a triple node Tri(i,j,k) is

$$1 + Yao(n_j, \lceil \frac{|C_j|}{B_{fan}} \rceil, |C_j|) + Yao(n_j, \|Tri(i, j, k)\|, |C_j|)$$

The case for backward navigation is similar.

4.4.3 Update Cost

Suppose there is an update on objects in class C_k which causes an update operation $\delta(k, k+1)$ (either insertion or deletion) on a triple node Tri(X,k,k+1), $0 \leq X < k$, with n_k number of identifiers of distinct objects of class C_k and n_{k+1} number of identifiers of distinct objects of class C_{k+1} in the tuples in $\delta(k, k+1)$.

The cost of updating the triple node Tri(X,k,k+1) is

$$\left[1 + Yao(n_k, \lceil \frac{|C_k|}{B_{fan}} \rceil, |C_k|) + Yao(n_k, \|Tri(X, k, k+1)\|, |C_k|)\right]$$

$$+Yao(n_k, \|Tri(X,k,k+1)\|, |C_k|)$$

In this equation, the first term (inside the square brackets) is the cost of navigating n_k tuples from the triple node, and the last term is the cost of write-back operations for updating the related pages.

Updates in triple node Tri(X,k,k+1) will generate a set of update operations $\delta(X,k+1)$. The number of identifiers of distinct objects of class C_X in $\delta(X,k+1)$ is estimated to be $bwd(X,k,n_k)$ and the number of identifiers of distinct objects of class C_{k+1} in $\delta(X,k+1)$ is n_{k+1}.

Similarly, the cost of updating the triple node Tri(k,k+1,Y), for $Y > k+1$ is

$$(1 + Yao(n_{k+1}, \lceil\frac{|C_{k+1}|}{B_{fan}}\rceil, |C_{k+1}|) + Yao(n_{k+1}, \|Tri(k,k+1,Y)\|, |C_{k+1}|))$$

$$+Yao(n_{k+1}, \|Tri(k,k+1,Y)\|, |C_{k+1}|)$$

and the set of update operations $\delta(k,Y)$ generated will consists of $fwd(k+1,Y,n_{k+1})$ number of identifiers of distinct objects of class C_Y and n_k number of identifiers of distinct objects of class C_k.

In general, for an update operation $\delta(i,j)$, with n_i number of identifiers of distinct objects of class C_i and n_j number of identifiers of distinct objects of class C_j in the tuples in $\delta(i,j)$,

the cost of updating the triple node Tri(X,i,j) is

$$(1 + Yao(n_i, \lceil\frac{|C_i|}{B_{fan}}\rceil, |C_i|) + Yao(n_i, \|Tri(X,i,j)\|, |C_i|))$$

$$+Yao(n_i, \|Tri(X,i,j)\|, |C_i|)$$

the cost of updating the triple node Tri(i,j,X) is

$$(1 + Yao(n_j, \lceil\frac{|C_j|}{B_{fan}}\rceil, |C_j|) + Yao(n_j, \|Tri(i,j,X)\|, |C_j|))$$

$$+Yao(n_j, \|Tri(i,j,X)\|, |C_j|)$$

class	C_0	C_1	C_2	C_3	C_4	C_5	C_6
$\|C_i\|$	2000	1500	2000	1800	2000	1500	1800
fan_i	0.8s	0.9s	1.0s	1.1s	0.8s	1.2s	1.8s
class		C'_1				C'_5	C'_6
$\|C_i\|$		1500				1500	1800
fan_i		0.9s				1.2s	1.8s
s	selectivity factor (variable)						

Table 4.2: Application-specific Parameters

the cost for updating the triple node Tri(⊥,i,j) is

$$1 + Yao(n_i, \lceil\frac{|C_i|}{B_{fan}}\rceil, |C_i|) + 2 \times Yao(n_i, \|Tri(\bot, i, j)\|, |C_i|)$$

the cost for updating the triple node Tri(i,j,⊤) is

$$1 + Yao(n_j, \lceil\frac{|C_j|}{B_{fan}}\rceil, |C_j|) + 2 \times Yao(n_j, \|Tri(i, j, \top)\|, |C_j|)$$

4.5 Evaluation

In this section, we demonstrate the cost estimates for a few indexing techniques including Access Support Relations, the Join Index Hierarchies, and the Triple Node Hierarchy. Our sample aggregation hierarchy is shown in Figure 4.2. The running example tries to support both forward and backward navigations between frequently referenced classes includes C_0 and C_6 ($Q^{0,6}$), C_0 and C_4 ($Q^{0,4}$), and $C_{1'}$ and $C_{6'}$ ($Q^{1',6'}$).

Two different types of Access Support Relations are compared in our cost model : the Full Access Support Relations, and the decomposed Access Support Relations . Three types of Join Index Hierarchies are also compared : Complete, Partial and Based. [1]

Application-specific parameters are shown in Table4.2. Two Full Access Support Relations (labeled ASR) are constructed to support navigations. Five

[1]see section 3.6

Access Support Relations (labeled SASR) including $\{ASR^{0,2}$, $ASR^{2,4}$, $ASR^{4,6}$, $ASR^{1',4}$, $ASR^{4,6'}\}$ are decomposed according to the rules on sharing of ASR[16]. The Partial Join Index Hierarchy structure maintains the set of target nodes {JI(0,4), JI(0,6), JI(1',6')} for efficient navigation. The Partial Join Index Hierarchy consists of four auxiliary join index nodes { JI(0,2), JI(2,4), JI(4,6), JI(1',4), JI(4,6') }. The Triple Node Hierarchy maintains six triple nodes { Tri(0,4,$\top$), Tri($\perp$,0,4), Tri(0,6,$\top$), Tri($\perp$,0,6), Tri(1',6',$\top$), Tri($\perp$,1',6') }. The Triple Node Hierarchy is constructed as shown in Figure 4.6.

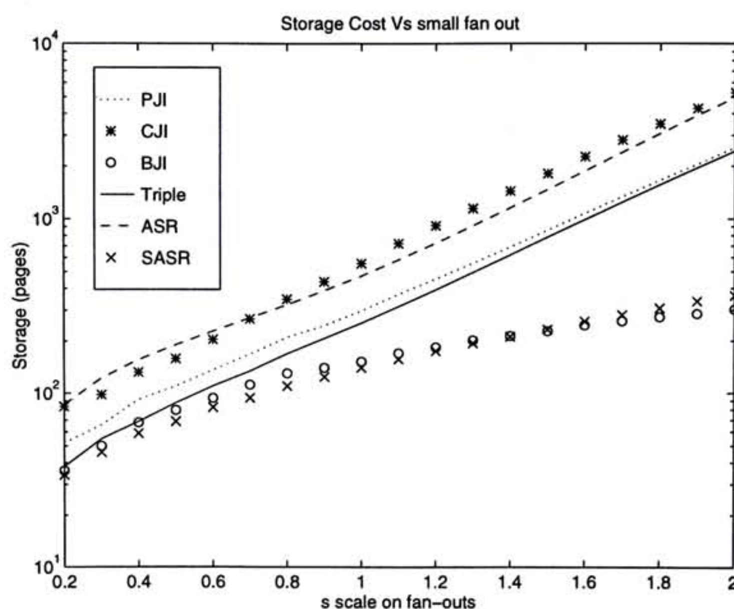


Figure 4.9: Storage Vs small fanout

The fan-out (join selectivity) among classes is critical to the constructions of join index nodes, triple nodes and Access Support Relations. The storage costs of the six index structures against various fan-out factors are shown in Figure 4.9.

In general, the storage costs increase as the fan-out factor increases. When the fan-out factor is small, the Triple Node Hierarchy, decomposed ASR, and the Based Join Index Hierarchy incur smaller storage costs than the others. Obviously, the costs of shared ASR and the Based Join Index Hierarchy increase more slowly than the others as the fan-out factor increases. This is due to their

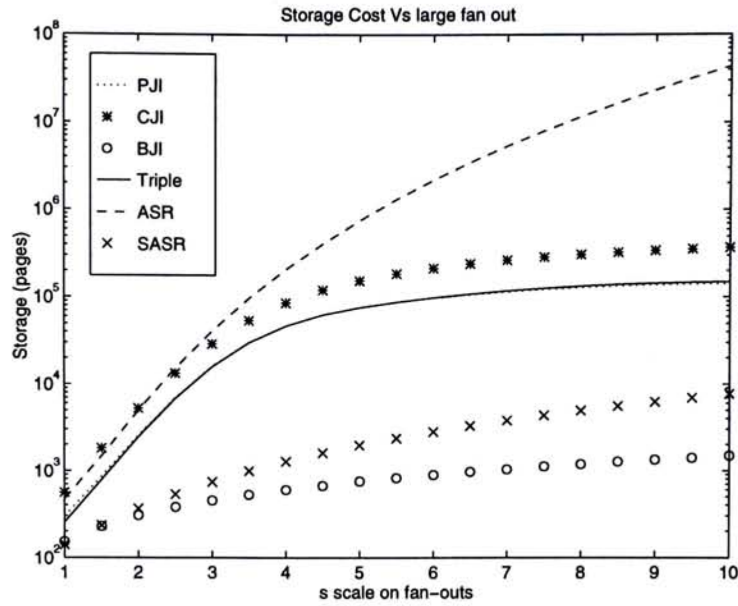


Figure 4.10: Storage Vs large fanout

simple structures. Figure 4.10 shows the rapid increase of the storage costs of the index structures under large fan-out factors.

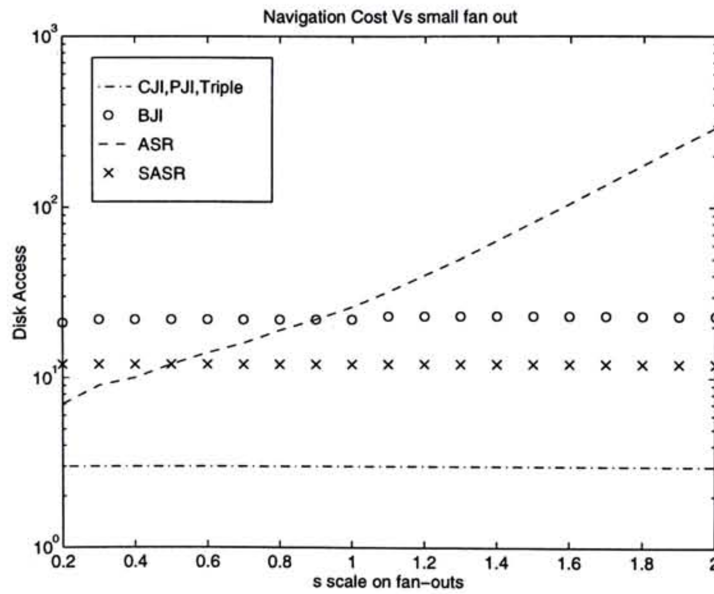


Figure 4.11: Navigation Cost Vs small fanout

Figure 4.11 shows the navigation costs of the six index structures. Here, we consider only the frequently referenced navigations , i.e. $(Q^{0,6}, Q^{0,4}, Q^{1',6'})$ with the same weight. We also assume that both the forward and backward navigations share the same weight. The navigation starts by only one object.

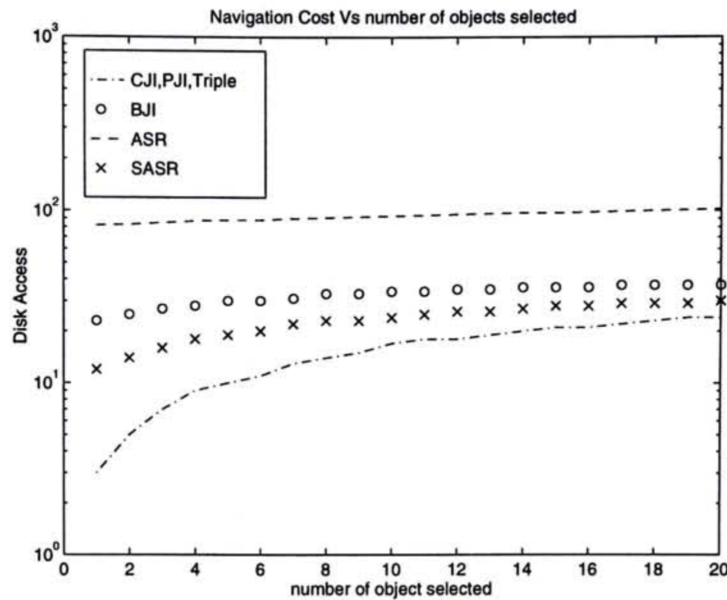


Figure 4.12: Navigation Cost Vs objects selected

As the Complete Join Index Hierarchy, the Partial Join Index Hierarchy and the Triple Node Hierarchy support navigations by maintaining direct mapping between frequently referenced object pairs, referenced navigations in Complete Join Index Hierarchy, Index Hierarchy and Triple Node Hierarchy are from the target nodes, therefore the navigation costs of the three index structures are the same. In fact, the navigation cost of the Complete Join Index is slightly lower than the other two because it provides full support for non-frequent navigations, which are considered as rare cases. For Full ASR, the navigations on $Q^{0,6}$ and $Q^{1',6'}$ are effective, because the indexes are clustered. However, the backward navigation cost on $Q^{0,4}$ requires a higher cost because $Q^{0,4}$ is not indexed and searching through the whole ASR is required. For the Base Join Index Hierarchy and decomposed ASR, a higher navigation cost is required because extra join operations are required. Figure 4.12 shows the navigation costs when the navigation starts from more than one object, and the selectivity factor is assumed to be 1.5. Obviously, the Triple Node Hierarchy, the Complete Join Index Hierarchy and the Partial Join Index Hierarchy perform better in navigation operations.

Figure 4.13 shows the update costs of the six index structures. We assume

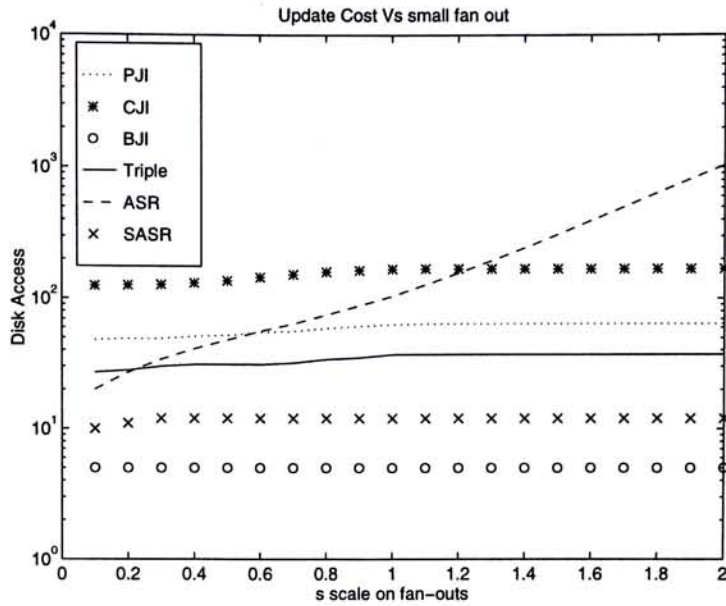


Figure 4.13: Update Cost Vs small fanout

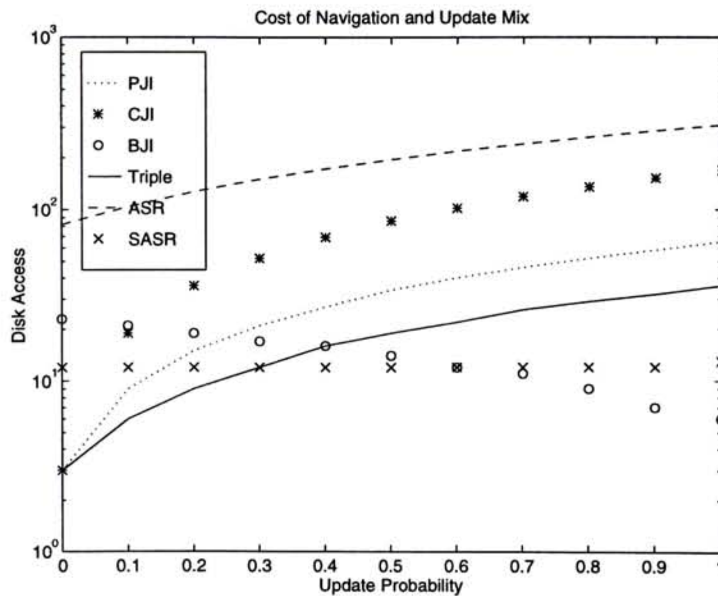


Figure 4.14: Navigation Update Mix (fanout set to 1.5)

that the update probability of each class is the same. We also assume that the update operation starts with only one relation. It is obvious that the update costs of decomposed ASR and the Based Join Index Hierarchy are lower because of their simple structures. The Triple Node Hierarchy has a lower update cost than the Complete and Partial Join Index Hierarchies as well as Full ASR. These results match our expectation as each time when update is propagated to higher

level nodes in the Join Index Hierarchy, extra join operations are required. These costs, however, are saved in the Triple Node Hierarchy.

Figure 4.14 describes the cost of navigation and update operation mix of the six index structures. The total cost is defined as $(1-p)\times$ Navigation Cost$+p\times$ Update Cost, where p is the update probability. The update probability =0.6 means that there are 60% updates and 40% navigations The fan-out factor is set to be 1.5. It can be shown that under a low update probability, the Triple Node Hierarchy outperforms the other methods.

4.6 Summary

In this chapter, we have presented an indexing structure for object-oriented databases, called the Triple Node Hierarchy. The Triple Node Hierarchy supports both forward and backward navigations among objects and classes along different path expressions. With suitable configuration, the cost on overlapping path expressions can be reduced. We have developed an analytical cost model and the simulation results have shown that the Triple Node Hierarchy performs better then other methods in object-oriented query processing, especially when the effect of overlapping path expressions is taken into account.

Chapter 5

Triple Node Hierarchy in Both Aggregation and Inheritance Hierarchies

5.1 Introduction

Structural indexing techniques discussed so far can be classified into techniques that provide support for nested predicates against nested attributes of an object (i.e. aggregation hierarchy) and into those that support queries issued against an inheritance hierarchy.

In an aggregation hierarchy, an attribute A_i of any class is logically an attribute of the root class, that is, the attribute A_i is a nested attribute of the root class. This nested attribute is inherited to the subclasses of the root class. Therefore, we need an indexing structure to support queries issued against the nested attribute in the inheritance hierarchy.

Previous indexing techniques have good performance in supporting queries issued against either an inheritance hierarchy or an aggregation hierarchy. However, few of them show how to support an integration of the inheritance hierarchy

and the aggregation hierarchy.

In the previous chapter, we have proposed the Triple Node Hierarchy method which supports fast navigations among objects and classes along path expressions. Forward object references from a set of objects of class C_j to objects of class C_k are supported by the triple node Tri($\perp$,j,k), while backward object references is supported by the triple node Tri(j,k,$\top$). The intermediate objects are maintained by the corresponding Triple Node Hierarchy.

In this chapter, we show how we can provide efficient support for queries against the integration of the inheritance hierarchy and the aggregation hierarchy.

5.2 Preliminaries

The Triple Node Hierarchy method presented in the previous chapter is designed for the support of aggregation hierarchy, i.e., navigations through a sequence of object classes via their attribute relationships. In a schema path which involves inheritance hierarchies, if a set of subclasses associated with the same higher-level class has similar kinds of attributes, it could be beneficial to construct one combined triple node for all classes instead of a large number of small triple nodes for each class in the inheritance hierarchy.

In this way, the tuples in the triple node Tri(i,j,k) may include OIDs of the instances from classes C_i, C_j and C_k, and all their subclasses which inherit the nested attribute from the superclasses. For the triple nodes Tri(i,j,$\top$) and Tri($\perp$,i,j) which are used to support navigations, we may either construct a combined triple node for all classes or a large number of small triple nodes for each class in the inheritance hierarchy.

For queries targeted on all classes in the inheritance hierarchy rooted at class C_i, the construction of a B^+-tree on the combined triple nodes Tri(i,n,$\top$) can

provide efficient support. We note, however, as the tuples in these nodes contain OIDs of instances of all classes in the inheritance hierarchy, this simple B^+-tree may not be able to provide efficient support for queries that target on one class.

If we construct several small triple nodes Tri(i_1,n,$\top$), Tri(i_2,n,$\top$),... for each class in the inheritance hierarchy instead of a combined one, the support for queries that are targeted on a single class is efficient. Moreover, queries targeted on all classes in the inheritance hierarchy may require navigation on several small triple nodes, which may be costly.

We note that there have been several indexing structures proposed in the literature which are able to solve this conflicting requirement and to provide efficient support for queries against an inheritance hierarchy. Examples are CH-trees, H-trees, hcC-trees, CG-trees and CD-trees. It is therefore interesting to integrate these indexing techniques with the Triple Node Hierarchy as a solution for indexing in an integration of both inheritance hierarchy and aggregation hierarchy.

We have investigated the integration of the Triple Node Hierarchy with the CH-tree. We choose CH-tree because it has a simple structure. Other methods, on the other hand, employ more complex data structures. Moreover, these methods have been proved by experiments that they have better performance than the CH-tree. We believe that the integration of the Triple Node Hierarchy with any one of these methods can achieve much better performance.

5.3 Class-Hierarchy Tree

Class-Hierarchy tree (CH-tree) was proposed by Kim et al. [20]. It has a simple structure based on the B^+-tree. As subclasses inherit attributes from their superclass, it is possible to maintain an index on the common attribute for all classes in an inheritance hierarchy. The domain of an attribute may be either a

primitive class or a non-primitive class. Thus, the key values in the index can be either some primitive values or the OIDs of the instances of the domain class. Each entry contains the OIDs of the instances of any class in the inheritance hierarchy.

The structure of a non-leaf node of a CH-tree is similar to that of a B^{+}-tree while the structure of leaf nodes of a CH-tree has a different structure.

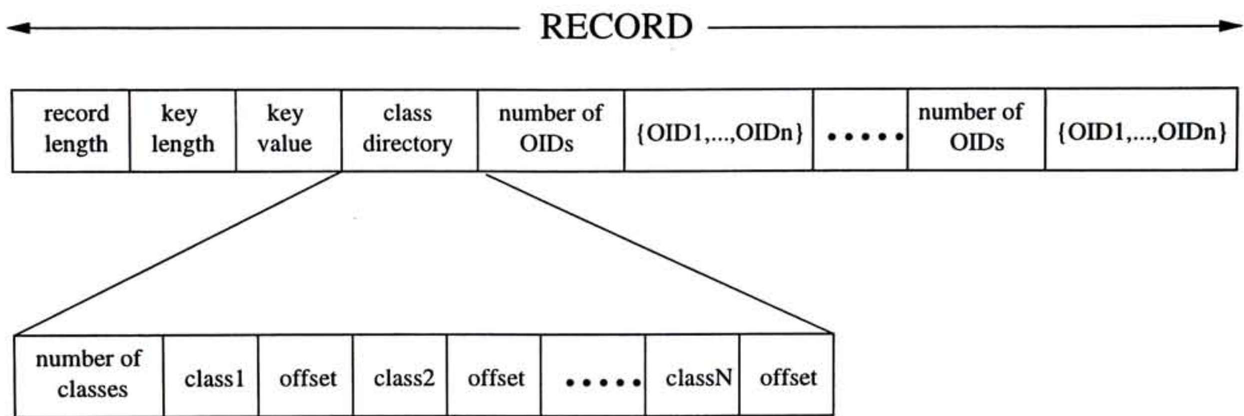


Figure 5.1: A leaf node record in CH-tree

A leaf node of a CH-tree consists of (1) a key value and (2) a key directory. For each class in the inheritance hierarchy, a CH-tree leaf node also consists of (3) a list of OIDs of instances of the class that hold the key value in the indexed attribute and (4) the number of elements in the OIDs list. The key directory contains an entry for each class that has instances with the key value in the indexed attribute. An entry for a class consists of the class identifier and the offset in the index record at which the list of OIDs of the objects can be found. The leaf node of a CH-tree groups the list of OIDs for a key value in terms of the classes to which they belong. Figure 5.1 shows an example of a leaf node in a CH-tree.

5.4 The Nested CH-tree

A Nested CH-tree (nCH-tree) is an CH-tree which is indexed on a nested attribute. However, the intermediate objects are supported by the Triple Node Hierarchy. The nCH-tree supports a fast evaluation of queries issued against a class or all classes in the inheritance hierarchy by a nested attribute, in the scope of a path expression in an aggregation hierarchy ending with that indexed attribute. Update operations do not require object traversals. Instead, they are propagated from the Triple Node Hierarchy. Note that a CH-tree is a special case of a Nested CH-tree. In a CH-tree the indexed attribute is a non-nested attribute of the root class of an inheritance hierarchy and thus the tree does not require the support of the Triple Node Hierarchy.

5.4.1 Construction

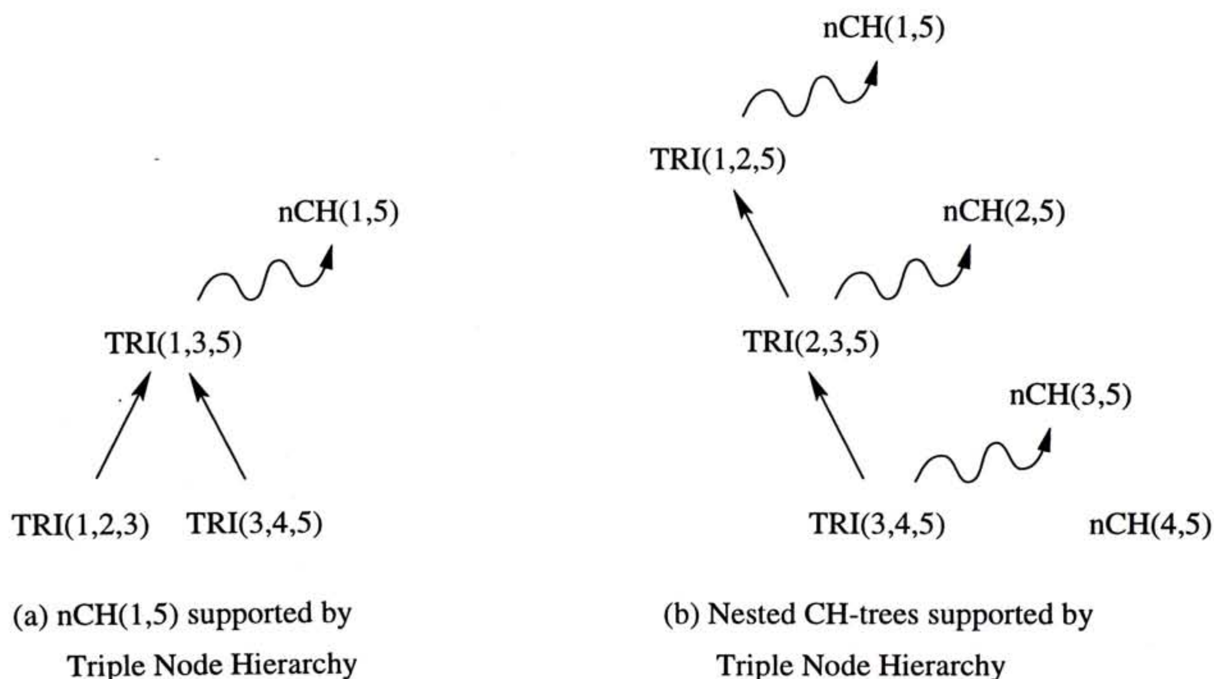


Figure 5.2: nCH-tree organizations

Let nCH-tree(i,j) denotes a nested CH-tree on an inheritance hierarchy rooted at class C_i indexed on the nested attribute A_j. Given a path expression $P =$

$C_1.A_1.A_2...A_n$, to construct a nCH-tree(1,n) is the same as constructing a Triple Node Hierarchy supporting the triple node Tri(1,n,$\top$). Moreover, the triple node Tri(1,n,$\top$) is organized as a CH-tree indexed on the values of A_n. Figure 5.2a shows the index organization of a nCH-tree (nCH-tree(1,5)) supported by the corresponding Triple Node Hierarchy. Figure 5.2b shows the index organization of four nCH-trees, including nCH-tree(1,5), nCH-tree(2,5), nCH-tree(3,5) and nCH-tree(4,5), supported by the corresponding Triple Node Hierarchy.

5.4.2 Retrieval

A retrieval operation performed in a nCH-tree is similar to that in a CH-tree.

In the nCH-tree(i,j), a predicate against the indexed attribute A_j on a single class is evaluated as follows. Let C be the class against which the predicate is issued. The index is scanned to find the leaf node record with the key value satisfying the predicate. Then the key directory is accessed to determine the offset in the index record where the list of OIDs of the instances of C is located. If there is no entry for class C, then there are no instances of C satisfying the predicate. If the query is against more than one class in the indexed inheritance hierarchy rooted at C_i, the predicate is processed in the same way, except that the lookup in the key directory is executed for each class involved in the query. Therefore, this access mechanism can be used when a query is applied only to some (not necessarily all) classes in the indexed graph.

5.4.3 Update

Update operation is propagated by the Triple Node Hierarchy to the nCH-tree. For a nCH-tree(i,j), an update in the attribute relationship within the scope of the nCH-tree may be reflected by an update operation $\delta(i,j)$ propagated by the Triple Node Hierarchy.

System Parameters	semantics, derivation/default
n	length of access path
P	net size of pages, which is set to 4096
$OIDL$	size of object identifiers, default is 8
kl	key length =2
of	offset length =2
tnc	size of the counter in a tuple of a triple node, =4
$type$	size of the TYPE flag in a tuple of a triple node, =1
tn	size of a tuple in a triple node, $=OIDL \times 2 + type + tnc$
PP	size of page pointer, =4
α	average page occupancy factor, =70%
B_{fan}	fan out of the B^+ tree, $B_{fan} = \lceil \frac{P \times \alpha}{PP+OIDL} \rceil$
N_v	number of contiguous key values in the range specified for a given query
N_e	number of entries in a leaf page
h_{ch}	Internal height of the nCH-tree
$h_{i,j,k}$	Internal height of the triple node Tri(i,j,k)
n_l	number of nodes at level l in a B^+-tree

Table 5.1: System Parameters

5.5 Cost Model

An analytical model is constructed to study the performance of the nCH-tree. The study focuses on several crucial performance measurements, including the cost of navigations and the cost of updates. The cost model constructed in the previous chapter considers only a simple case of an aggregation hierarchy. Here such a cost model is extended to cover inheritance hierarchies.

Table 5.1 lists some system parameters used in the cost analysis. Table 5.2 and Table 5.3 list the database parameters. These parameters describe the classes in a path expression $P = C_1.A_1.A_2...A_n$. $C_{i,1}$ denotes the root class of the inheritance hierarchy having position i in the path expression. The details of the estimation of some of these parameters are in Appendix A.

Logical data parameters	semantics, derivation/default
nc_i	Number of classes in the inheritance sub-hierarchy rooted at class $C_{i,1}$, $1 \leq i \leq n$
$D_{i,j}$	Number of distinct values for attribute A_i of class $C_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq nc_i$
D_i	Number of distinct values for attribute A_i for all instances in the inheritance sub-hierarchy rooted at class $C_{i,1}$; $D_i = \sum_{j=1}^{nc_j} D_{i,j}$
$N_{i,j}$	Number of instances of class $C_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq nc_i$
Nh_i	Number of members of class $C_{i,1}$, $1 \leq i \leq n$; $Nh_i = \sum_{j=1}^{nc_i} N_{i,j}$
$fan_{i,j}$	Average number of references to members of class $C_{i+1,1}$, contained in the attribute A_i for an instance of class $C_{i,j}$, $1 \leq i \leq n$ and $1 \leq j \leq nc_i$. The average is computed with respect to all the instances of class $C_{i,j}$, $1 \leq i \leq n$ and $1 \leq j \leq nc_i$.
fan_i	Average number of references to members of class $C_{i+1,1}$, contained in the attribute A_i of a member of class $C_{i,1}$, $1 \leq i \leq n$. The average is computed with respect to all the members of class $C_{i,1}$, $1 \leq i \leq n$. The difference between this parameter and the previous is that this parameter is obtained as the average evaluated on all members of a class hierarchy, while in the previous the average is for each class. Note: $fan_i = \frac{\sum_{j=1}^{nc_i} d_{i,j} * fan_{i,j}}{\sum_{j=1}^{nc_i} d_{i,j}}$
$d_{i,j}$	Number of instances of class $C_{i,j}$, having a value different from Null for attribute A_i, $1 \leq i \leq n$ and $1 \leq j \leq nc_i$
d_i	Number of members of class $C_{i,1}$, having a value different form Null for attribute A_i, $1 \leq i \leq n$; $d_i = \sum_{j=1}^{nc_i} d_{i,j}$
$k_{i,j}$	Number of instances of class $C_{i,j}$, having the same value for attribute A_i, $1 \leq i \leq n$ and $1 \leq j \leq nc_i$; $k_{i,j} = \frac{d_{i,j} * fan_{i,j}}{D_{i,j}}$
kh_i	Number of members of class $C_{i,1}$, having the same value for attribute A_i, $1 \leq i \leq n$ Note: $kh_i = \frac{d_i * fan_i}{D_i}$

Table 5.2: Database Parameters

Derived Parameters	semantics, derivation/default
$\overline{k}_{i,j}$	Number of instances of class $C_{i,j}$ having the same value for the nested attribute A_n $1 \leq i \leq n$, $1 \leq j \leq nc_i$
$\overline{fan}_{i,j}$	Average number of objects held in the nested attribute A_n of an instance of the class $C_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq nc_i$. The average is computed with respect to all the instances of class $C_{i,j}$, $1 \leq i \leq n$ and $1 \leq j \leq nc_i$
$RefByh(i,y,k)$	average number of values contained in the nested attribute A_y for a set of k members of class $C_{i,1}$, with $1 \leq i \leq y \leq n$ and $1 \leq k \leq D_y$.
$Refh(i,y,k)$	average number of members of class $C_{i,1}$ having as value of the nested attribute A_y a value in a set of k elements, with $1 \leq i \leq y \leq n$.
$Def_{A_n}(i,j)$	Number of instances of class $C_{i,j}$ having at least one value, different from Null, for the nested attribute A_n, $1 \leq i \leq n$, $1 \leq j \leq nc_i$

Table 5.3: Derived Parameters

5.5.1 Assumptions

1. All key values have the same length.
2. The number of classes containing instances with the indexed key value is the same for each key value.
3. Key values are uniformly distributed among instances.
4. The root class of a class hierarchy for which a nCH-tree is maintained is also the class against which a query is directed.
5. Each non-leaf (non-root) index node has the same fanout.
6. The cardinality of a class in a class hierarchy is independent of the cardinality of any of its super/subclasses. e.g. abstract class has no instances at all.

These assumptions are commonly found in analytical models for database access structures. Note: distribution of key values across classes of a class hierarchy is

important.

5.5.2 Storage

The average size of a primary (leaf-node) record of the nested CH-tree is:

$$XCH = OIDL * \left\lceil \sum_{i=1}^{nc_1} \overline{k}_{1,i} \right\rceil + kl + (OIDL + of) * nc_1$$

The number of tuples in a triple node Tri(i,j,k) is

$$|Tri(i,j,k)| = Nh_j * RefByh(i,j,1) + Nh_j * Refh(j,k,1)$$

The number of pages needed to store a triple node is

$$\|Tri(i,j,k)\| = \lceil \frac{tn \times |Tri(i,j,k)|}{P \times \alpha} \rceil$$

5.5.3 Query Cost

The average number of pages required to store a leaf-node record is given by:

$$np_{ch} = \left\lceil \frac{XCH}{p} \right\rceil$$

Let c be the number of classes targeted by a query, i.e. c represents the number of classes whose instances must be retrieved from the index lookup. The average number of pages accessed for an index lookup for a point query is given by:

$$C_{ch}(retrieve_{point}) = \begin{cases} h_{ch} + 1 & , \quad XCH \leq P \\ h_{ch} + npa_{ch} + [1 - \frac{npa_{ch}}{np_{ch}}] & , \quad XCH > P \end{cases}$$

where $npa_{ch} = Yao(c, np_{ch}, nc_1)$. npa_{ch} represents the number of pages that are surely accessed, and $1 - \frac{npa_{ch}}{np_{ch}}$ is the probability of accessing another page in the case when the class directory is not stored in one of the npa_{ch} pages. If the target of the query is the set of all classes in the inheritance hierarchy, we have $c = nc_1$.

The number of entries in a leaf page of a nCH-tree is :

$$N_e = \lfloor \frac{P}{XCH} \rfloor$$

the average number of pages accessed for an index lookup for a range query having N_v number of contiguous values in the range is given by:

$$C_{ch}(retrieve_{range}) = \begin{cases} h_{ch} + \lceil \frac{N_v}{N_e} \rceil & , \quad XCH \leq P \\ h_{ch} + (npa_{ch} + [1 - \frac{npa_{ch}}{np_{ch}}]) * N_v & , \quad XCH > P \end{cases}$$

5.5.4 Update Cost

Consider two types of update operations in the relationship between two classes:

- Insert an instance $O_{i,t}$ into the attribute A_{i-1} of another instance $O_{i-1,s}$
- Delete an instance $O_{i,t}$ from the attribute A_{i-1} of another instance $O_{i-1,s}$

In both cases, an update operation $\delta(i-1,i)$ will be generated (either insertion or deletion). This update operation will be propagated from the Triple Node Hierarchy to the nCH-tree.

There are two different cases to be considered.

1. The object $O_{i,t}$ has a Null value for the nested attribute A_n;
2. The object $O_{i,t}$ has values different from Null for the nested attribute A_n;

For case 1, no leaf node records in the nCH-tree are updated. Only the Triple Node Hierarchy is updated and the cost is estimated as follows[1]:

For an update operation $\delta(i,j)$, with n_i number of identifiers of distinct objects of class C_i and n_j number of identifiers of distinct objects of class C_j in the tuples in $\delta(i,j)$,

the cost of updating the triple node Tri(X,i,j) is

$$C_{tri_retrive}(n_i, X, i, j) = VisBtree\,(n_i, Nh_i, ht_{X,i,j}) + Yao(n_i, \|Tri(X,i,j)\|\,, Nh_i)$$

[1] see sections 4.3.3 and 4.4.3 for the details of the update propagation

the cost of updating the triple node Tri(i,j,X) is given by:

$$C_{tri_retrive}(n_j, i, j, X) = VisBtree\,(n_j, Nh_j, ht_{i,j,X}) + Yao(n_j, \|Tri(i,j,X)\|, Nh_j)$$

where $VisBtree(k,n,h)$ denotes the cost of the batch scanning of a B^+-tree of height h with a total number of n keys to retrieve k keys. Suppose that the leaf node records may be stored in only one page, then the value of $VisBtree(k,n,h)$ is given by :

$$VisBtree(k,n,h) = \sum_{l=1}^{h} npa_l$$

where npa_l denotes the number of page accessed at level l in a B^+-tree.

$$npa_l = \begin{cases} Yao(k, n_l, n) & , \quad l = h \\ Yao(npa_{l+1}, n_l, n_{l+1}) & , \quad l < h \end{cases}$$

and n_l denotes the number of nodes at level l in a B^+-tree, with $n_h = \lceil \frac{n}{B_fan} \rceil$ and $n_l = \lceil \frac{n_{l+1}}{B_fan} \rceil, 1 \leq l < h$.

For case 2, both the nCH-tree and the Triple Node Hierarchy are updated. The number of updated leaf pages of the nCH-tree is

$$DNP_{ch}(i,t) = \begin{cases} Yao(\overline{fan}_{i,t}, LP_{ch}, D_n) & , \quad XCH \leq P \\ \overline{fan}_{i,t} * \left(NPC_{ch} + \left[1 - \frac{NPC_{ch}}{np_{ch}}\right]\right) & , \quad else \end{cases}$$

where $NPC_{ch} = \lceil np_{ch}/nc_1 \rceil$.

The update cost of the nCH-tree is given by:

$$CH_{update}(i,t) = 2 * DNP_{ch}(i,t) + VisBtree\left(\overline{fan}_{i,t}, D_n, h_{ch}\right)$$

The probability that case 1 occurs is:

$$P1 = \frac{N_{i,t} - Def_{A_n}(i,t)}{N_{i,t}}$$

The total update cost can be estimated by merging case 1 and case 2.

5.6 Evaluation

We have conducted a large number of simulation experiments on the basis of the mathematical cost model for the evaluation of the performance of the nCH-tree. We compare the nCH-tree with the nested-inherited index (NIX) [5]. NIX was proposed by Bertino that provides an efficient evaluation of nested predicates for both queries having as a target a single class and queries having as a target any number of classes in a given inheritance graph. As both the nCH-tree and the NIX can provide an integrated treatment of indexing in the framework of both aggregation and inheritance hierarchy, it is appropriate to compare the performance of the two index structures. The cost model for the NIX is given in Appendix B.

The system parameters in Table 5.1 are set according to common implementations of the B^+-tree index. We variate the parameters in Table 5.2 for all classes in the path expression.

5.6.1 Storage Cost

To evaluate the storage requirements of the nCH-tree and the NIX organization, we have performed a set of experiments by varying the number of classes, and hence the total number of instances, and the fanout parameter.

We consider a path expression $P = C_1.A_1.A_2.A_3.A_4$ with length $= 4$. We assume that there are four inheritance hierarchies associated with each class in the path expression, and each of these hierarchies carries the same number of classes and there are 10000 instances in each class. The parameters d, D, and N are set to 10000 for all classes. The key value is the nested attribute A_4, which is an attribute of class $C_{4,1}$.

As the NIX organization provides support for every class in the scope of the path expression, in order to have a fair comparison, we constructed four

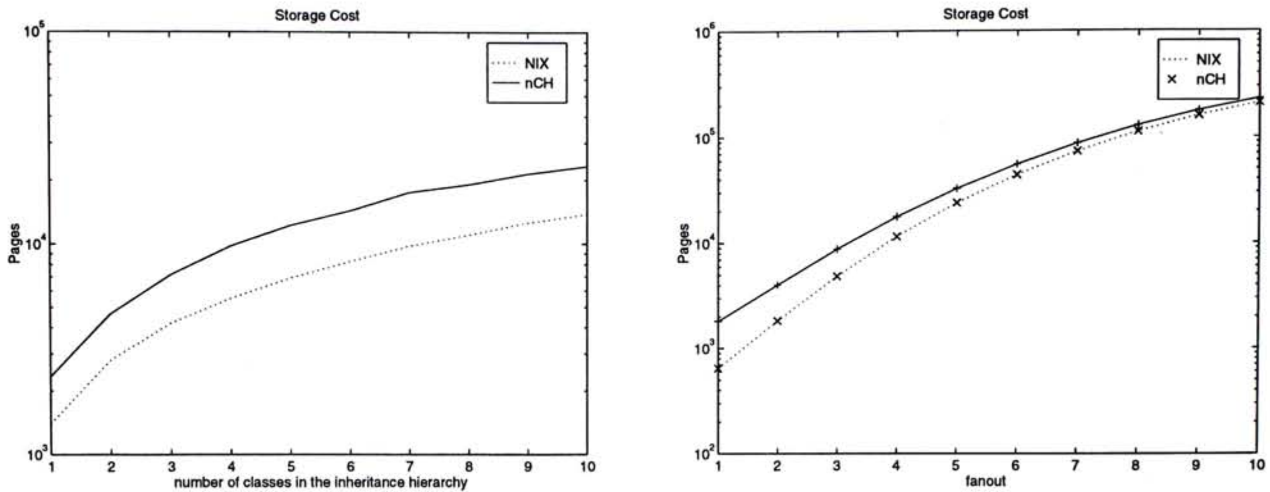


Figure 5.3: Storage Cost

nCH-trees, one for each class in the path expression. However, as the nCH-trees support only frequently referenced navigations, the storage cost of nCH-trees may be actually lower than the results reported here; this happens when some of these nCH-trees are considered as non-frequently navigations.

Figure 5.3a shows the storage cost of the nCH-tree and NIX organization against the number of classes in each inheritance hierarchy. The fanout parameter is set to 1 for all classes, except for the classes rooted at $C_{4,1}$, which is set to 20. As the number of classes in each inheritance hierarchy increases, the total number of object instances also increases, and thus more space is needed. This factor is reflected in the storage cost graph.

Figure 5.3b shows the storage cost of the nCH-tree and NIX organization against the fanout parameter. The number of classes in each inheritance hierarchy is set to 3, and the fanout is the same for all classes. Obviously, the costs of both structures increase as the fanout parameter increases.

Although the results show that the nCH-tree has a higher storage cost than the NIX organization, this may not be crucial because large capacity storage devices are widely available in the market. Therefore, it may be preferable to privilege organizations which provide good performance, even if they have large storage requirements.

5.6.2 Query Cost

In the query efficiency study, we have performed several simulation experiments by varying different factors that influence the performance of the indexing structures.

These factors include

1. The number of classes in the inheritance hierarchies rooted at the classes in a path.

2. The query range.

3. The path lengths.

4. The position of large fanout.

In the experiments reported here, the parameters d, D, and N are set to 10000 for all classes. We consider a path expression with length set to 4. In oder to compare the nCH-tree with the NIX organization, we constructed four nCH-trees, one for each class in the path. The nCH-tree and the NIX are indexed on the nested attribute A_4, which is an attribute of class $C_{4,1}$. We assume that there are four inheritance hierarchies in the path expression, each having the same number of classes. The retrieval cost is calculated by averaging the cost on retrieval operations on each of the four classes.

In the first experiment, we measured the retrieval cost against the number of classes in the inheritance hierarchies rooted at the classes in a path. The value of the fanout parameter fan is set to 1 for all classes except those rooted at $C_{4,1}$, which is set to 20. The range queries cover 0.001% of the available key values. Fig5.4 shows the average retrieval cost of the nCH-tree and NIX organization against the number of classes in the inheritance hierarchies, for queries targeted on a single class and on all classes in the inheritance hierarchy. Obviously, the nCH-tree has a lower retrieval cost in both cases. The retrieval cost of the NIX

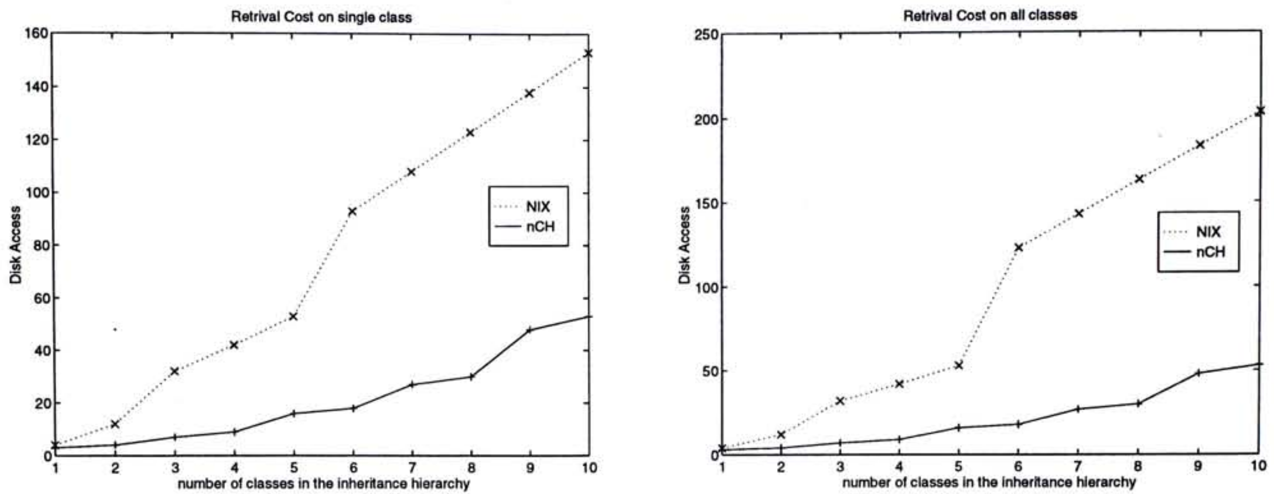


Figure 5.4: Retrieval Cost Vs number of classes in the inheritance hierarchies

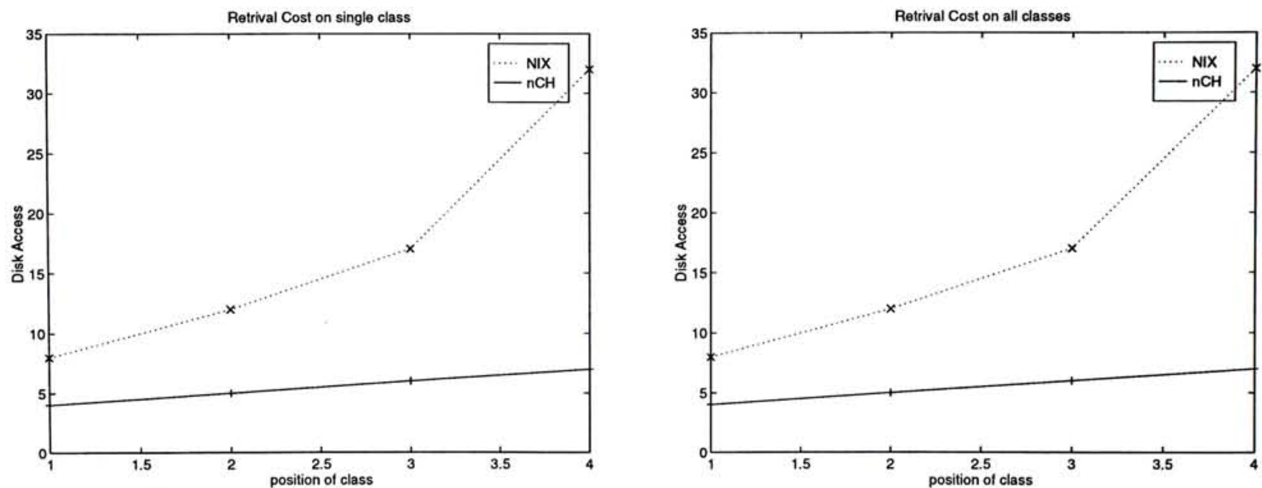


Figure 5.5: Retrieval Cost Vs position of class having large fanout

increases faster than that of the nCH-tree. It is because of the rapid growth of the size of the leaf node record in the NIX.

In the second experiment, we measured the retrieval cost against the position of the classes with the large fanout parameter fan. The value of the fanout parameter fan is set to 1 for all classes except the classes rooted at the position being measure, which is set to 20. The range queries cover 0.001% of the available key values.

Fig5.5 shows the average retrieval cost of the nCH-tree and NIX organization against the number of classes in the inheritance hierarchies, for queries targeted on a single class and on all classes in the inheritance hierarchy. The number

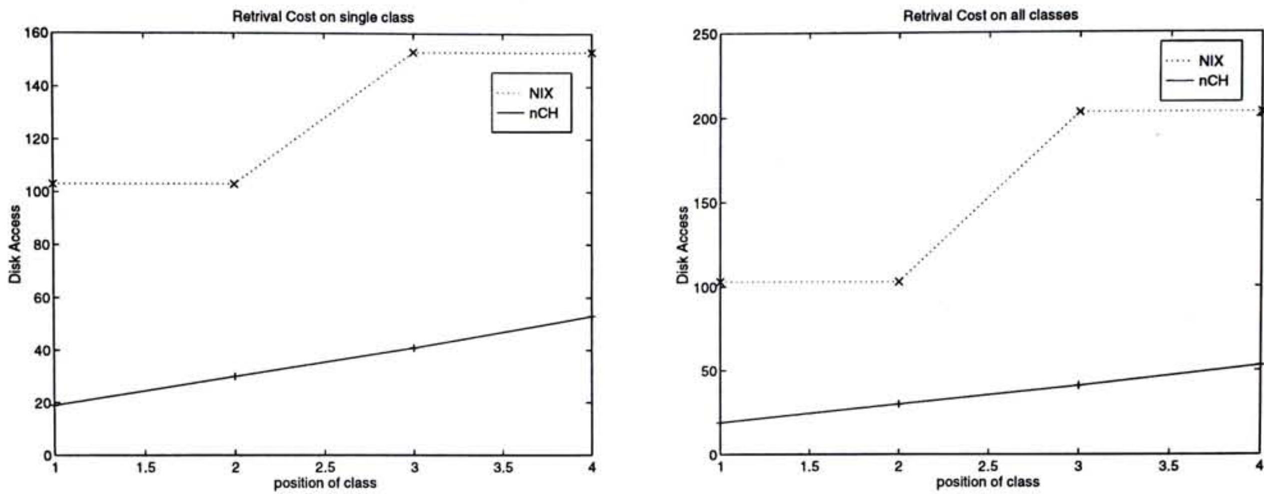


Figure 5.6: Retrieval Cost Vs position of class having large fanout

of classes in each inheritance hierarchies is set to 3. Obviously, the nCH-tree has a lower retrieval cost in both cases. The retrieval cost of the nCH-tree and NIX increase as the position of classes having a large fanout shifts towards the indexed attribute. This is because for an object with a large fanout, the instances that reference this object will also have a large fanout. Therefore, as the position of classes having a large fanout shifts towards the indexed attribute, the number of objects having the same key value increases. The growth of the size of the leaf node record incurres more on the retrieval cost.

Fig5.6 shows the rapid growth of the retrieval cost of the NIX organization, when the number of classes in each inheritance hierarchy is set to 10.

In the third experiment, we measured the retrieval cost against the range of key values covered by the queries. The values of the fanout parameter fan is set to 1 for all classes except the classes rooted at $C_{4,1}$, which is set to 20. The range of key values covered by the queries is varied to cover from 0.001% of the available key values to 0.010%.

Fig5.7 shows the average retrieval cost of the nCH-tree and NIX organization against the range of queries targeted on a single class and on all classes in the inheritance hierarchy. The number of classes in each inheritance hierarchies is set to 3. Obviously, the nCH-tree has a lower retrieval cost in both cases. The

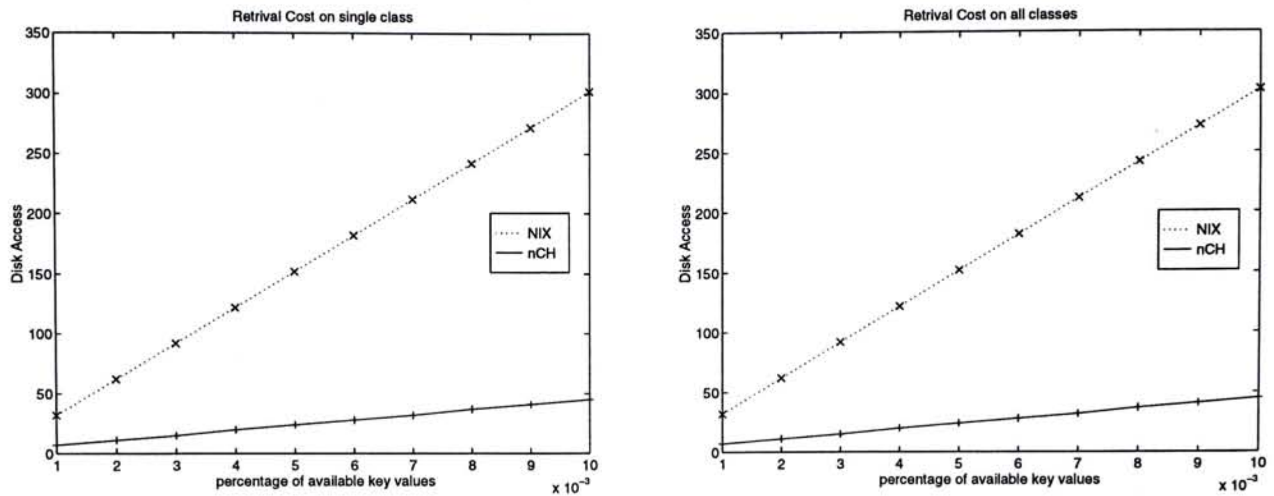


Figure 5.7: Retrieval Cost Vs range of key values covered by the queries

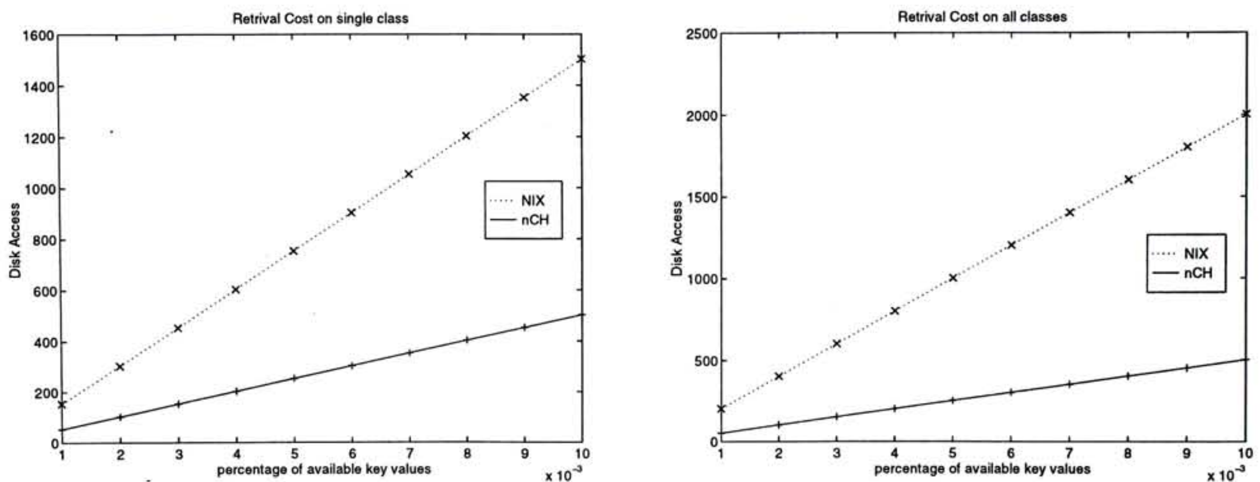


Figure 5.8: Retrieval Cost Vs range of key values covered by the queries

retrieval cost of the NIX increases faster than the nCH-tree because the size of the leaf node records of the NIX is greater than that of the nCH-tree, which does not favor range search.

Fig5.8 shows the rapid growth of retrieval cost of the NIX organization, when the number of classes in each inheritance hierarchies are set to 10.

In the forth experiment, we measure the retrieval cost against the length of the path expression. The values of the fanout parameter fan is set to 1 for all classes except the classes rooted at the last class of the path expression, i.e. $C_{n,1}$, which is set to 20. The range of key values covered by the queries is set to 0.001% of the available key values.

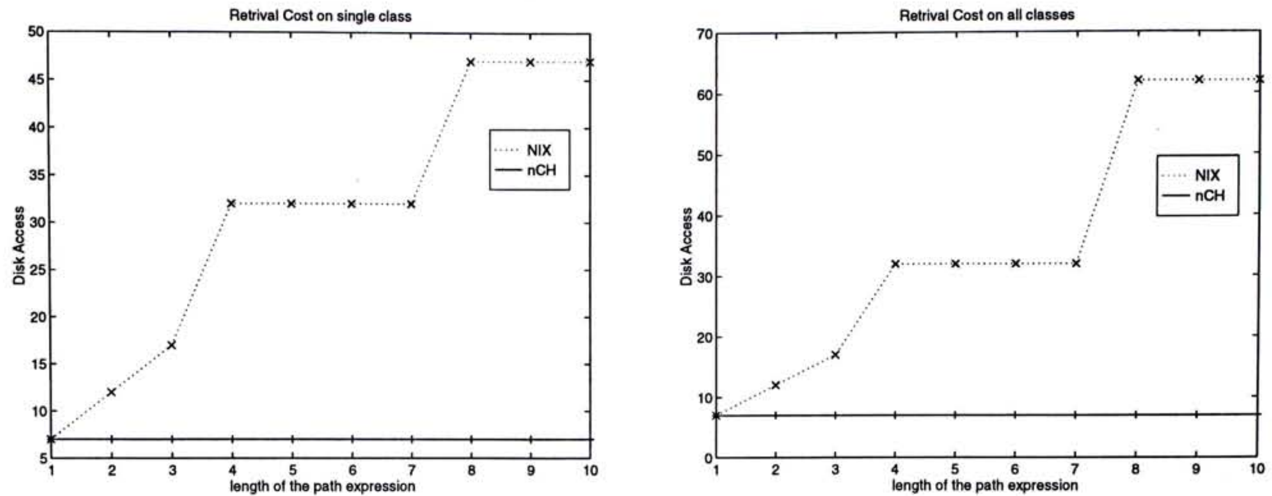


Figure 5.9: Average Retrieval Cost Vs path length

Fig5.9 shows the average retrieval cost of the nCH-tree and NIX organization against the length of the path expression. The number of classes in each inheritance hierarchies is set to 3. Increasing the length of the path does not affect the number of classes in each inheritance hierarchy, but the number of inheritance hierarchies increases. The retrieval cost of the nCH-tree keeps very low in both cases while the retrieval cost of the NIX increases as the path length increases. This is because as the path length increase, the size of the leaf node records of the NIX also increases to store the additional information on classes. The nCH-trees, on the other hand, stores only the information on a single inheritance hierarchy, and thus, it is not affected by additional classes in different positions.

Conclusions on retrieval cost

From the experiments that have been performed, the results match our expectation, that is, navigations by the nCH-tree structures offer better performance than the NIX organization in most cases. The nCH-tree stores only the OIDs of the target classes in the inheritance hierarchy with the indexed key value, while the intermediate objects that are irrelevant to the navigation operations are stored separately in the Triple Node Hierarchy. This simple structure allows

the nCH-tree to have a smaller leaf node size and thus more leaf nodes can be stored in a page. This small leaf node size is beneficial for range queries which involves the retrieval of records for contiguous key value. Moreover, the small leaf node size also lowers the height of the B^+-tree in some cases. These factors make the nCH-tree achieve better performance. The NIX organization, on the other hand, stores all OIDs of all classes within the scope of the path expression in the leaf node records. The size of a leaf node record in the NIX could be quite large in some cases. Moreover, the height of the B^+-tree structure in the primary record of the NIX organization may be higher than that of nCH-tree in some cases. The performance on range queries is then affected.

We note that the cost of NIX depends on the total number of classes within the scope of the path expression. The cost of nCH-tree, however, depends on only the total number of classes in the target inheritance hierarchy.

These evaluations are also valid for point queries. In that case, the retrieval cost depends mainly on the height of the B^+-tree structures associated with the index organization.

5.6.3 Update Cost

To demonstrate the update cost of the nCH-tree and NIX organization, we have performed a set of experiments in which the path expression has a length of 4. We assume that each class in the path expression has two subclasses, i.e. each inheritance hierarchy carries 3 classes. There are 10000 instances in each class and the parameters d, D, and N are set to 10000 for all classes. The key value is the nested attribute A_4 which is an attribute of class $C_{4,1}$. The running example tries to index on the root class with the nested attribute at the end of the path.

Figure 5.10 shows the average update costs of the two index structures. We assume that the update probability of each class is the same. We also assume that the update operation starts by only one relation. In Figure 5.10, the upper

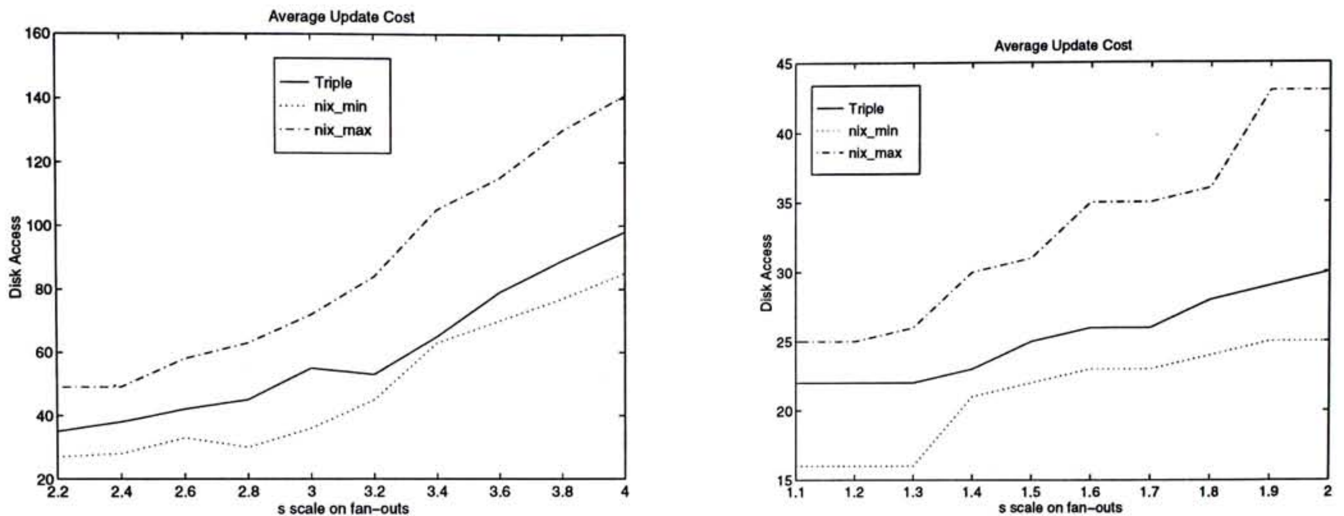


Figure 5.10: Update Cost

bound and the lower bound of the update cost of NIX are estimated. It is shown that the nCH-tree has an update cost in between the boundaries of the NIX organization. In general, the NIX has a lower update cost when the update is performed on the starting of the path. As the updated classes shift towards the ending of the path, the update cost increases. This is because the auxiliary index is need to be accessed. As the path length gets longer, it can be expected that the update cost of the NIX will be higher. The nCH-tree does not show this problem.

5.7 Summary

In this chapter, we have addressed the need for index organizations which support both aggregation and inheritance hierarchies and have presented the nCH-tree which is an integration of the Triple Node Hierarchy and the CH-tree. The nCH-tree supports a fast evaluation of queries on the nested attribute issued against a class or all classes in the inheritance hierarchy. We have developed an analytical cost model and our simulation results show that navigations through the nCH-tree offer a better retrieval performance in most cases. Update operations propagated from the Triple Node Hierarchy require a reasonable cost which

is highly dependent on the frequently referenced object topologies. In general, if several nCH-trees are constructed on overlapping path expressions, update operations supported by the Triple Node Hierarchy offer a better performance.

The nCH-tree shows a possibility of integrating the Triple Node Hierarchy method with other indexing methods. It would be interesting to integrate the Triple Node Hierarchy with, say the CD-tree, to construct a nested CD-tree organization and to compare the performance. We believe that the construction of different nested tree structures supported by the Triple Node Hierarchy could be a promising candidate towards the indexing support on query processing.

Chapter 6

Decomposition of Path Expressions

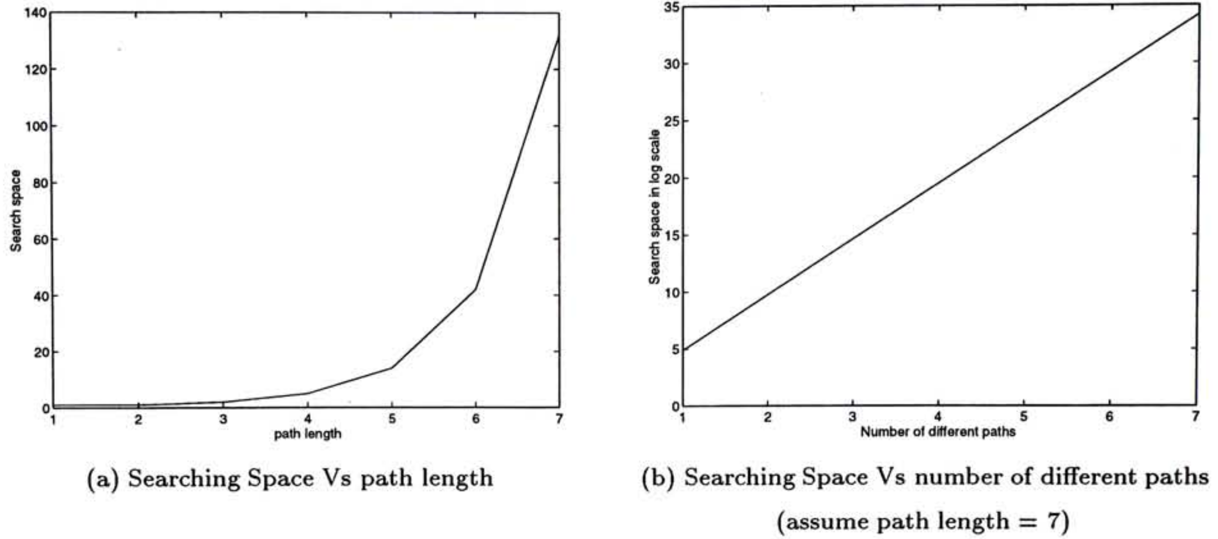
6.1 Introduction

As the set of triple nodes maintained in a Triple Node Hierarchy is determined by the decomposition sequences of the path expressions it covers, it is reasonable to select the best decomposition sequences of these path expressions so that an optimal configuration can be determined with minimum costs.

In [33], an algorithm was proposed to derive an optimal configuration which minimizes (1) the total storage costs; and (2) the total number of update operations in an update propagation. The set of overlapping path expressions is modeled by a set of target nodes, and the decomposition of these target nodes is modeled by a set of auxiliary nodes. This algorithm selects the minimum set of auxiliary nodes by searching all possible combinations of nodes, i.e. by testing all possible decomposition sequences. This algorithm is shown in Appendix C.

The algorithm can determine the optimal configuration because it tests all possibilities. We note, however, that the performance of this algorithm can be quite bad as the total number of possible combination increases. Appendix D

shows the total number of different combinations on a path expression with a path length n is $\frac{1}{n} *_{2n-2} C_{n-1}$. A lot of computing power and storage are required to execute the algorithm when the path length is long.



(a) Searching Space Vs path length

(b) Searching Space Vs number of different paths (assume path length = 7)

Figure 6.1: Performance measurement of original algorithm

Figure 6.1a shows that the search space (i.e. the total number of different possible combinations to be tested) increases exponentially as the path length increases. Figure 6.1b shows that the search space increases when the number of different path expressions increases. In general, when k different path expressions in the aggregation hierarchy are taken into account, the total number of different combinations becomes:

$$\prod_{i=1}^{k} \frac{1}{n_i} *_{2n_i-2} C_{n_i-1}$$

We note that the length of the path expressions may be very long in some cases. The number of overlapping path expressions, on the other hand, is limited to a small number because only path expressions with frequently referenced object pairs are maintained. Based on this observation, we present an alternative algorithm which can be used to configure the decomposition of the path expressions when the original algorithm does not perform well.

6.2 Configuration on Path Expressions

6.2.1 Single Path Expression

To configure the decomposition of a single path expression, a simple solution is to balance the length of the two new subpath expressions.

Rule 1: balance-load-decomposition

Given a path expression $P(i,k) = C_i.A_i...A_{k-1}$, the two subpath expressions generated are $P(i,j)$ and $P(j,k)$ where $i < j < k$ and the value of j is:

$$j = \begin{cases} \frac{i+k}{2} & , \quad i + kisodd \\ \frac{i+k-1}{2} & , \quad i + kiseven \end{cases}$$

With this rule, the length of the two new subpath expressions is balanced and the changes between any two logically connected objects in the path expression will have similar effect and update costs.

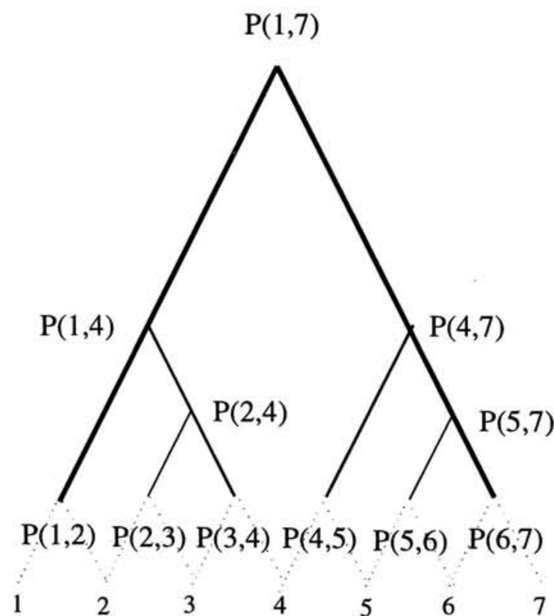


Figure 6.2: Decomposition Graph for P(1,7)

By applying the balance-load-decomposition recursively, we can obtain the decomposition graph. For example, if we apply the rule to the path expression

P(1,7), the decomposition graph is described as:
P(1,7) is decomposed at 4 into P(1,4) and P(4,7),
P(1,4) is decomposed at 2 into P(1,2) and P(2,4),
P(2,4) is decomposed at 3 into P(2,3) and P(3,4),
P(4,7) is decomposed at 5 into P(4,5) and P(5,7), and
P(5,7) is decomposed at 6 into P(5,6) and P(6,7)
as shown in figure 6.2.

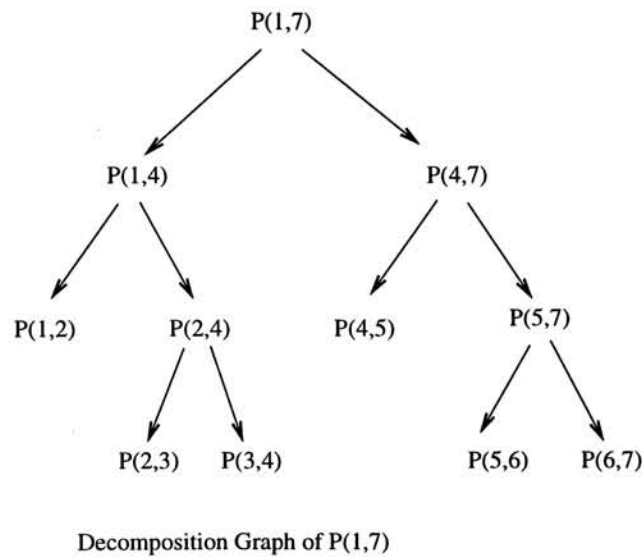


Figure 6.3: Decomposition Graph for P(1,7)

6.2.2 Overlapping Path Expressions

There may be cases where a set of frequently referenced object pairs are supported. These object pairs lie on a set of path expressions in the aggregation hierarchy which may be overlapped. We have demonstrated how to generate the decomposition sequence of a single path expression in the previous section. For overlapping path expressions, we need another approach to determine the decomposition sequence of these path expressions.

Suppose we have to support two frequently referenced object pairs which lie on two overlapping path expressions $P(i,j)$ and $P(m,n)$. There are two cases:

1. $P(i,j)$ includes $P(m,n)$:

   The path expression $P(m,n)$ is part of another path expression $P(i,j)$, where $i \leq m$ and $j \geq n$.

   Rule 2: inclusive-decomposition

   $P(i,j)$ needs to be decomposed into 3 partitions, $P(i,m)$, $P(m,n)$, $P(n,j)$ to share the subpath expression $P(m,n)$.

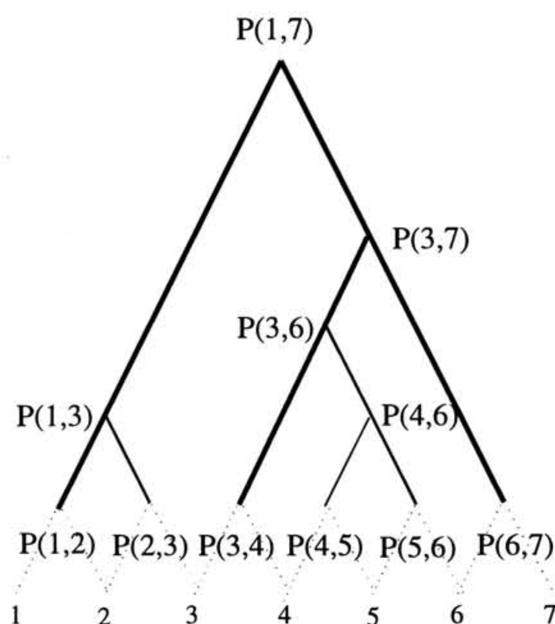


Figure 6.4: Decomposition Graph for {P(1,7), P(3,6)}

   Figure 6.4 shows the decomposition graph for $P(1,7)$ and $P(3,6)$. $P(1,7)$ includes $P(3,6)$ and thus $P(1,7)$ is decomposed at at 3 into $P(1,3)$ and $P(3,7)$. $P(3,7)$ in turn includes $P(3,6)$. Finally, $P(3,6)$ is decomposed at 4 into $P(3,4)$ and $P(4,6)$ according to rule 1.

   Note that the path will be decomposed into two partitions when $i = m$ or $n = j$.

2. $P(i,j)$ intersects $P(m,n)$ at $P(a,b)$:

   There exists a subpath expression $P(a,b)$ such that $P(m,n)$ includes $P(a,b)$ and $P(i,j)$ includes $P(a,b)$.

Rule 3: intersect-decomposition

The path $P(i,j)$ needs to be decomposed into 3 partitions, $P(i,a)$, $P(a,b)$ and $P(b,j)$. $P(m,n)$ needs to be decomposed into 3 partitions, $P(m,a)$, $P(a,b)$ and $P(b,n)$ in order to share the path $P(a,b)$. Each of the subpaths can be further decomposed.

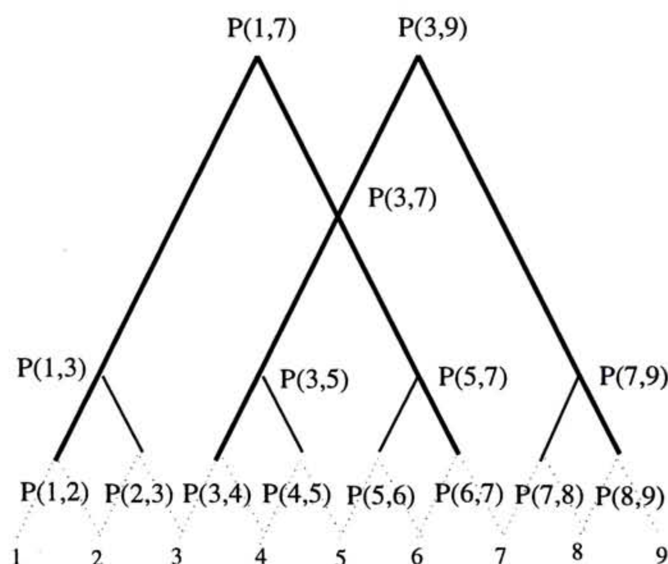


Figure 6.5: Decomposition Graph for {P(1,7), P(3,9)}

Figure 6.5 shows the decomposition graph for $P(1,7)$ and $P(3,9)$. $P(1,7)$ intersects $P(3,9)$ at $P(3,7)$. $P(1,7)$ is therefore decomposed at 3 into $P(1,3)$ and $P(3,7)$ while $P(3,9)$ is decomposed at 7 into $P(3,7)$ and $P(7,9)$. $P(3,7)$ is then decomposed according to rule 1.

Note that inclusive-decomposition is a special case of intersect-decomposition when $m = a$ and $n = b$.

6.3 New Algorithm

We have presented two decomposition rules to determine the decomposition sequence of a pair of overlapping path expressions. For a set of overlapping

---

Path_Decompose()

Input: a set of path expressions $\{P(m_0, n_0), P(m_1, n_1), ..., P(m_k, n_k)\}$
Output: a decomposition graph for the set of path expressions

1. For each of the path expressions, find the set of their decomposition points

```
for i = 1 to k
   for j = 1 to k ≠ i
      if P(m_i, n_i) includes P(m_j, n_j)
         mark m_j and n_j to denote a decomposition point
      else if P(m_i, n_i) intersects with P(m_j, n_j) at P(m_x, n_x)
         mark m_x and n_x to denote a decomposition point
   end for
end for
```

2. For each decomposition point in the set of path expressions, generate a set of subpaths. The subpaths are a decomposition of the target path according to the decomposition point determined. For example, if P(i,j) has a decomposition point at k, then the set of subpaths will include the set P(i,k),P(k,j). Notice that any subpaths included in the set of path expressions will be removed from the set of subpaths. If there is an empty set resulted from this removal, return the empty set.

For each of the path expressions in the set, if it has l sets of subpaths, make l copies of path expressions, and add each set of subpaths to a l copy to form l new sets.

This process repeats until no new decomposition point can be found. For all paths remained in the set, apply the balance-load decomposition to obtain a set of subpaths. The result is the decomposition graph for the set of path expressions with minimum costs.

Figure 6.6: Construction of Decomposition Graph for a set of Overlapping Path Expressions

path expressions, the decomposition sequence is complicated because each path expression may have different overlapping regions and thus may have more than one overlapping subpath expressions. Figure 6.6 outlines the algorithm used to determine the decomposition sequence.

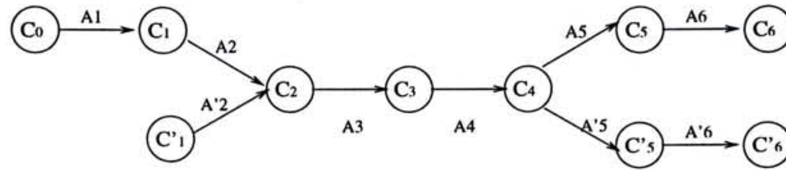


Figure 6.7: Overlapping Path Expressions

6.3.1 Example

We use an example to demonstrate how the algorithm works. Suppose we have an aggregation hierarchy as shown in figure6.7, and our target is to support forward and backward navigations between frequently referenced classes (C_0, C_6), (C_0, C_4) and (C'_1, C'_6). Therefore, the decomposition graph is determined as follows:

Initially, the set of path expressions is {{P(0,4), P(0,6), P(1,6')}}.

The path expression P(0,4) has 1 decomposition point which is at 2, and will be decomposed into {P(0,2),P(2,4)}.

The path expression P(0,6) has 2 decomposition points, at 2 and 4, and will be decomposed into either {P(0,2),P(2,6)} or {P(0,4),P(4,6)}. In the second set, P(0,4) is in the set of path expressions to be removed.

The path expression P(1',6') has 2 decomposition points at 2 and 4. and will be decomposed into either P(1',2),P(2,6') or P(1',4),P(4,6').

Therefore, there are totally $1\times2\times2 = 4$ possible decompositions:

1. {P(0,2),P(2,4),P(2,6),P(1',2),P(2,6')}

2. {P(0,2),P(2,4),P(2,6),P(1',4),P(4,6')}

3. {P(0,2),P(2,4),P(4,6),P(1',2),P(2,6')}

4. {P(0,2),P(2,4),P(4,6),P(1',4),P(4,6')}

In the first set P(2,6) has one decomposition point at 4 and P(2,6') has one decomposition point at 4. The resulting set of subpaths is {P(2,4),P(4,6),P(4,6')}

and P(2,4) will be removed. P(0,2) and P(2,4) are decomposed into {P(0,1),P(1,2)} and {P(2,3),P(3,4)} respectively by rule 1. Finally, P(4,6) and P(4,6') are decomposed into P(4,5),P(5,6) and P(4,5'),P(5',6') respectively by rule 1. The process stops at the point when no more decomposition points can be found in the above subpaths. The set generated is {P(0,2), P(2,4), P(2,6), P(1',2), P(2,6'), P(4,6), P(4,6'), P(0,1), P(1,2), P(2,3), P(3,4), P(4,5), P(5,6), P(4,5'), P(5',6')}

Similarly, we obtain the following sets of subpaths

{ P(0,2), P(2,4), P(2,6), P(1',2), P(2,6'), P(4,6), P(4,6'), P(0,1), P(1,2), P(2,3), P(3,4), P(4,5), P(5,6), P(4,5'), P(5',6') }

{ P(0,2), P(2,4), P(2,6), P(1',4), P(4,6'), P(0,1), P(1,2), P(2,3), P(3,4), P(4,6), P(1',2), P(4,5'), P(5',6'), P(4,5), P(5,6) }

{ P(0,2), P(2,4), P(4,6), P(1',2), P(2,6') P(0,1), P(1,2), P(2,3), P(3,4),P(4,5), P(5,6), P(4,6'), P(4,5'), P(5',6') }

{ P(0,2), P(2,4), P(4,6), P(1',4), P(4,6') P(0,1), P(1,2), P(2,3), P(3,4), P(4,5), P(5,6), P(1',2), P(4,5'), P(5',6') }

It is found that the fourth set is more preferable than others because it has a smaller number of join operation counts and has a lower update cost.

Thus, the decomposition graphs of the three path expressions are composed of:

P(0,6) decomposed at 4 into P(0,4) and P(4,6),

P(0,4) decomposed at 2 into P(0,2) and P(2,4),

P(0,2) decomposed at 1 into P(0,1) and P(1,2),

P(2,4) decomposed at 3 into P(2,3) and P(3,4),

P(4,6) decomposed at 5 into P(4,5) and P(5,6),

P(1',6') decomposed at 4 into P(1',4) and P(4,6'),

P(1',4) decomposed at 2 into P(1',2) and P(2,4), and

P(4,6') decomposed at 5' into P(4,5') and P(5',6').

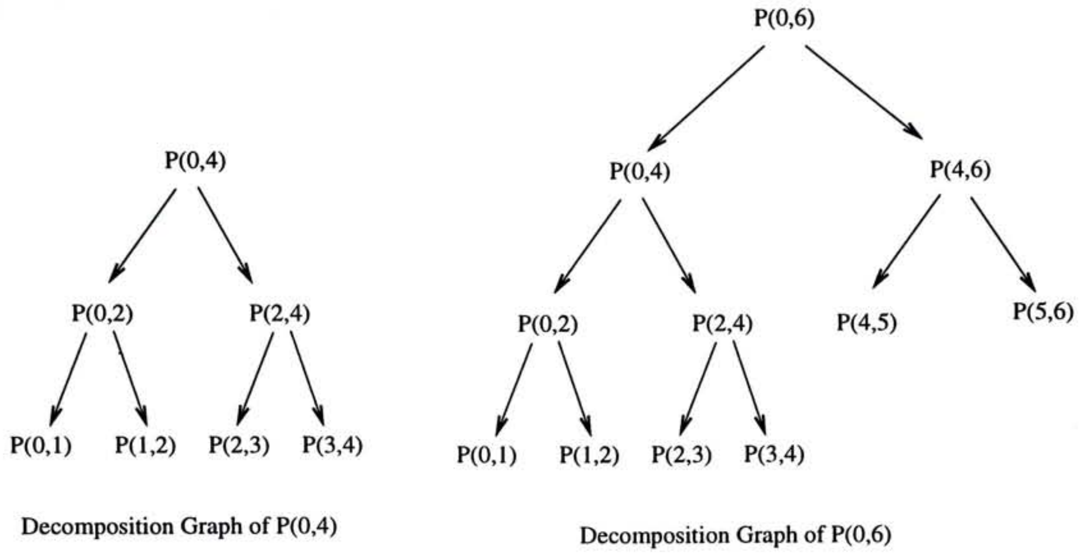


Figure 6.8: Decomposition Graph of P(0,4) and P(0,6)

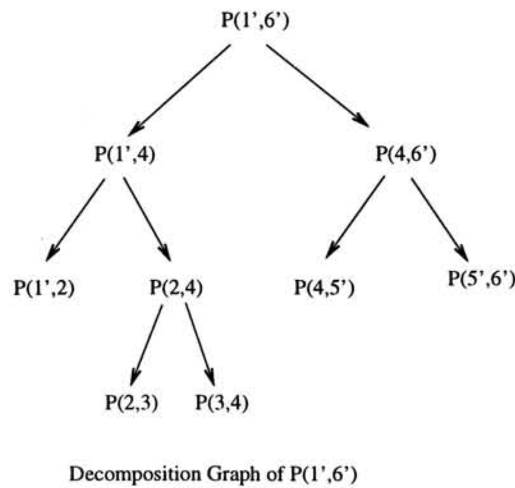
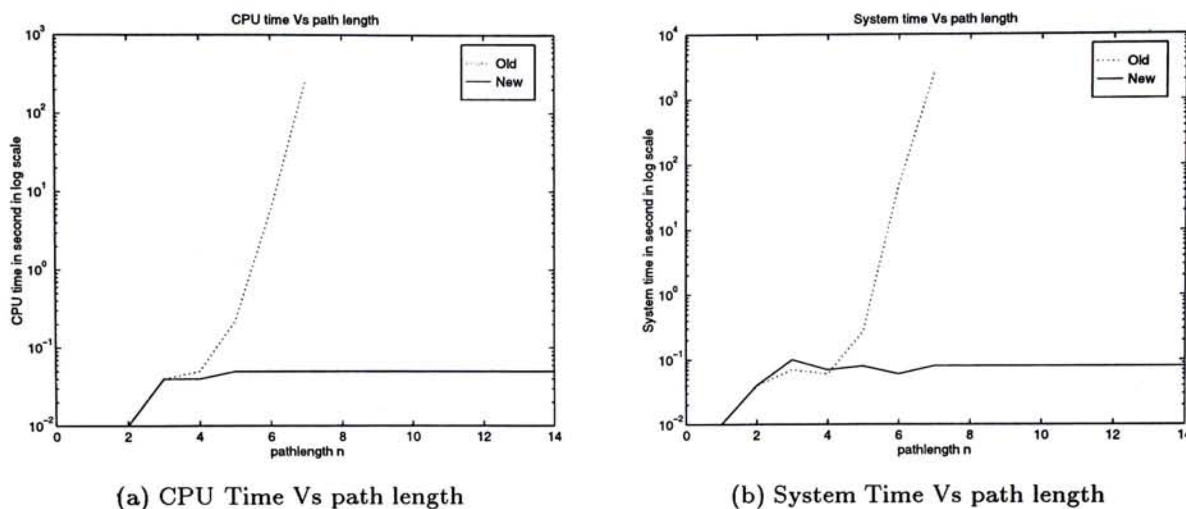


Figure 6.9: Decomposition Graph of P(1',6')

The decomposition graphs of the three path expressions are shown in figure 6.8a,figure 6.8b and figure 6.9. Note that figure 6.8a is part of figure 6.8b.

6.4 Evaluation



(a) CPU Time Vs path length

(b) System Time Vs path length

Figure 6.10: CPU Time and System Time Vs path length

In order to demonstrate the effectiveness and performance of the new algorithm, we performed a test on the two algorithms. We tried to find the auxiliary node sets for the set {P(0,n), P(1,n+1), P(2,n+2)} using the two algorithms. We implemented the two algorithms and executed in a SPARC20 machine. For the old algorithm, it has been shown that $(\frac{1}{n} *_{2n-2} C_{n-1})^3$ different combinations of nodes were searched.

We measured both the CPU time and the system time required to perform the task. The results are shown in figure 6.10a and 6.10b. When n is small(i.e. the paths are short) the system responds immediately in both algorithms. However, as n increases, the response time for the old algorithm increases exponentially. For the new algorithm, the response time is still fast. For $n = 7$, the system time for the old algorithm is 2648 seconds while the CPU time is 268 seconds. The large difference between the system time and the CPU time is caused by the large amount of data being kept in the virtual memory, and thus much swapping into and out of the memory is required.

Obviously, the new algorithm has a better performance over the old one, especially under long path lengths. The experiment has shown that in long path

expressions, the old algorithm performs poorly due to the large search space requirement. The experiment cannot be completed for the old algorithm for a large n (say for $n = 10$).

6.5 Summary

In this chapter, we have presented an algorithm which configures the decomposition of a set of path expressions and generates a decomposition graph supporting the construction of the Triple Node Hierarchy. The algorithm is derived based on the observation that the length of the path expressions may be very long in some cases while the number of overlapping path expressions is limited to a small number because only path expressions with frequently referenced object pairs are maintained. The algorithm can be used to configure the decomposition of the path expressions when performing tests on all possible decomposition sequences is not possible. Our analysis has shown that the algorithm performs well.

Chapter 7

Conclusion and Future Research

7.1 Conclusion

Object-Oriented Database systems provide powerful modeling facilities, but require efficient query evaluation to achieve high performance in advanced applications. In this thesis, we have investigated the influence of indexing structures on query optimization. As a result, several important issues have been identified, including the support for efficient navigation. The following research results constitute the major contributions of the thesis.

- Triple Node Hierarchy for fast navigations among objects and classes along path expressions.
- Configuration of the Triple Node Hierarchy on overlapping path expressions.
- Integration of the Triple Node Hierarchy with the CH-tree for fast navigations in both aggregation hierarchy and inheritance hierarchy.

7.2 Future Research

The work reported in this thesis can be extended along several directions.

First of all, the Triple Node Hierarchy can be combined with other indexing structures, such as hcC-trees, CG-trees, CD-trees, to enhance the abilities in supporting inheritance hierarchies. These structures have been proved by experiments that they have better performance than the CH-tree. Thus, it is expected that the integration of the Triple Node Hierarchy with any one of these methods can achieve much better performance. Moreover, some system parameters such as the fanout factors may impact on the costs of the indexing structures. Thus, an extensive comparison on the costs and performance among the combined organizations is essential. In addition, the combined organizations can be integrated with query optimization strategies to support efficient processing of complex queries containing several nested predicates. Furthermore, as the cost of the Triple Node Hierarchy may be reduced by selecting an optimal decomposition graph of path expressions, it is therefore necessary to develop effective algorithms to configure the splitting of long path expressions into several subpath expressions.

Appendix A

Evaluation of some Parameters in Chapter 5

The probability that a given O_{i+1}, member of class $C_{i+1,1}$ and belonging to the definition domain of attribute A_i of cardinality D_i, is not referenced by a given member of class $C_{i,1}$ is: $1 - (fan_i/D_i)$
if $kh_i > 1$, the probability that $Oi + 1$ is not referenced by any object belonging to a set of member of class $C_{i,1}$, with attribute A_i having a set of values $\{O^1_{i,j}, O^2_{i,j}, ..., O^k_{i,j}\}$ all different form Null is the following :

$$PrD(i,k) = \left(1 - \frac{fan_i}{D_i}\right)^k$$

if kh_i =1,

$$PrD'(i,k) = \prod_{y=0}^{k-1} \left(1 - \frac{fan_i}{(D_i - y * fan_i)}\right)$$

Let RefBy(i,s,y,k) ($0 \leq i \leq y \leq n$, $1 \leq s \leq nc_i$ and $1 \leq k \leq D_y$) denote the number of values contained in the nested attribute A_y for a set of k instances of class $C_{i,s}$.

$$RefBy(i,s,y,k) = \begin{cases} D_{i,s} * (1 - v(i,s) * Pr(i,s,k) - \\ (1 - v(i,s)) * Pr'(i,s,k)) & y = i \\ D_y * (1 - v'(i) * PrD(y, RefBy(i,s,y-1,k) * P_{A_y - 1}) - \\ (1 - v'(i)) * PrD'(y, RefBy(i,s,y-1,k) * P_{A_y} - 1)) & y > i \end{cases}$$

where

$$v(i,s) = \begin{cases} 1 & k_{i,s} > 1 \\ 0 & else \end{cases}$$

$$v'(i) = \begin{cases} 1 & kh_i > 1 \\ 0 & else \end{cases}$$

The number of values contained in the nested attribute A_y for a set of k members of class $C_{i,1}$ is

$$RefByh(i,y,k) = \begin{cases} D_i * (1 - v'(i) * PrD(i,k) - \\ (1 - v'(i)) * PrD'(i,k)) & y = i \\ D_y * (1 - v'(i) * PrD(y, RefByh(i,y-1,k) * P_{A_y - 1}) - \\ (1 - v'(i)) * PrD'(y, RefByh(i,y-1,k) * P_{A_y} - 1)) & y > i \end{cases}$$

Thus, the average number of objects held in the nested attribute A_n of an instance of the class $C_{i,j}$, $1 \le i \le n$, $1 \le j \le nc_i$ is:

$$\overline{fan}_{i,j} = RefBy(i,j,n,1)$$

For the formulations $\overline{k}_{i,j}$ and $Def_{A_n}(i,j)$, two other parameters must be derived; $Ref(i,j,y,k)$ and $Refh(i,y,k)$. These parameters represent, respectively, the average number of instances of class $C_{i,j}$ which have a value in a set of k elements ($1 \le i \le y \le n$ and $1 \le j \le nc_i$) as the value of the nested attribute A_y, and the average number of members of class $C_{i,1}$ which have a value in a set of k elements ($1 \le i \le y \le n$) as the value of the nested attribute A_y. The formulation of these parameters follows the same approach as the formulation of

the parameters $RefBy(i,j,y,k)$ and $RefByh(i,y,k)$. A detailed description of the formulations of these parameters is given in [4]. The following is obtained; $\overline{k}_{i,j} = Ref(i,j,n,1)$ and $Def_{A_n}(i,j) = Ref(i,j,n,D_n)$.

Appendix B

Cost Model for Nested-Inherited Index

Table B.1 lists the parameters that are used to derive the cost model for the NIX structure. The derivations for these parameter can be found in [4]. The cost model is quite similar to that given in [5].

B.1 Storage

The average size of a primary (leaf-node) record XNI is given by:

$$XNI = OIDL * \left[\sum_{i=1}^{n} \sum_{j=1}^{nc_i} \overline{k}_{i,j} \right] + kl + (OIDL + of + pp) * \left[\sum_{i=1}^{n} nc_i \right]$$

The average number of pages required to store a leaf-node record is

$$np = \left\lceil \frac{XNI}{p} \right\rceil$$

The average size of a 4-tuples of an auxiliary record associated with the class $C_{i,j}$ is

$$AUX_{i,j} = OIDL * (1 + kh_{i-1}) + pp * \overline{fan}_{i,j}$$

Derived Parameters	semantics, derivation/default
$\overline{k}_{i,j}$	Number of instances of class $C_{i,j}$ having the same value for the nested attribute A_n $1 \leq i \leq n$, $1 \leq j \leq nc_i$
$\overline{kh}_i$	Number of members class $C_{i,1}$ having the same value for the nested attribute A_n, $1 \leq i \leq n$
$\overline{k}^t(i,j)$	Number of instances of class $C_{i,j}$ having a given object as value of attributeA_t,$1 \leq i \leq n$,$1 \leq j \leq nc_i$ and $i \leq t < n$
$\overline{kh}_i^t$	Number of members of class $C_{i,1}$ having a given object as value of attributeA_t,$1 \leq i \leq n$, and $i \leq t < n$
$\overline{k}^t_{A_n}(i,j)$	number of instances of class $C_{i,j}$ having the same value for the nested attribute A_n and having a given object as value of attribute A_t, $1 \leq i \leq n$, $1 \leq j \leq nc_i$ and $i \leq t \leq n$
$\overline{fan}_{i,j}$	Average number of objects held in the nested attribute A_n of an instance of the class $C_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq nc_i$. The average is computed with respect to all the instances of class $C_{i,j}$, $1 \leq i \leq n$ and $1 \leq j \leq nc_i$
$\overline{fan}^t_{i,j}$	Average number of objects held as value of the nested attribute A_t of an instance of the class $C_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq nc_i$, $i \leq t < n$. The average is computed with respect to all the instances of class $C_{i,j}$, $1 \leq i \leq n$ and $1 \leq j \leq nc_i$
$RefVal_{A_n}(i,j,t)$	Number of instances of class $C_{i,j}$ that contain at least one value of S in the nested attribute A_n where S is a subset of the definition domain of the attribute A_n of cardinality t, $1 \leq i \leq n$, $1 \leq j \leq nc_i, t \geq 0$
$Ref_{out_k}(i,j,y,m)$	Number of instances of class $C_{i,j}$ that are roots of at least one instantiation *I of the path with the following characteristic. Let be* S a subset of *m elements of the definition domain of the attribute A_y and* O an object member of the class $C_{k,1}$ Then the instantiation *I must not contain* O and must have as last value an element of *S*, $1 \leq i \leq n$, $1 \leq j \leq nc_i$, $i \leq y \leq n$, $i \leq k \leq y$, $m \geq 0$. *Note that the instantiation* I may be also partial on condition that *I has such length as to contain an instance of $C_{i,j}$ and a member of $C_{y,1}$*
$Def_{A_n}(i,j)$	Number of instances of class $C_{i,j}$ having at least one value, different from Null, for the nested attribute A_n, $1 \leq i \leq n$, $1 \leq j \leq nc_i$
$Defh_{A_n}(i)$	Number of members of class $C_{i,1}$ having at least one value, different from Null, for the nested attribute A_n, $1 \leq i \leq n$

Table B.1: Database Parameters

The average size of a 4-tuples of an auxiliary record is

$$\overline{AUX} = \left\lceil \frac{\sum_{i=2}^{n} \sum_{j=1}^{nc_i} AUX_{i,j} * N_{i,j}}{\sum_{i=2}^{n} Nh_i} \right\rceil$$

The number of leaf pages of the auxiliary B^+-tree is

$$LP_A = \left\lceil \frac{\sum_{i=2}^{n} \sum_{j=1}^{nc_i} AUX_{i,j} * N_{i,j}}{P} \right\rceil$$

B.2 Query Cost

The average number of pages accessed for an point query targeted on a number c of classes is:

$$C(retrieve_{point}) = \begin{cases} h_p + 1 & , \quad XNP \leq P \\ = h_p + npa + [1 - \frac{npa}{np}] & , \quad XNP > P \end{cases}$$

where $npa = Yao(c, np, \sum_{i=1}^{n} nc_i)$

the average number of pages accessed for an index lookup for a range query having N_v number of contiguous values in the range is given by:

$$C(retrieve_{range}) = \begin{cases} h_p + \lceil \frac{N_v}{N_e} \rceil & , \quad XNI \leq P \\ h_p + (npa + [1 - \frac{npa}{np}]) * N_v & , \quad XNI > P \end{cases}$$

B.3 Update

Two types of update operations in the relationship between two classes is considered:

Insert $O_{i,t}$ into $O_{i-1,s}.A_{i-1}$

delete $O_{i,t}$ from $O_{i-1,s}.A_{i-1}$

If we insert an object $O_{i,t}$ as the attribute A_{i-1} of another object $O_{i-1,s}$, we may need to insert pointers in the auxilliary records which points to primary records. If we delete an object $O_{i,t}$ from the attribute A_{i-1} of another object

$O_{i-1,s}$, we may need to delete the pointers in the auxilliary records which points to primary records.

In both cases, the update cost is the same.

There are two cases for an update operation in NIX:

Case 1. The object $O_{i,t}$ has a Null value for the nested attribute A_n;

Case 2. The object $O_{i,t}$ has values different than Null for the nested attribute A_n;

For case 1, since no primary record is accessed, only tuple of $O_{i,t}$ is updated by removing $O_{i-1,s}$ from the parent list. The cost is given by: $C_{indef} = h_A + 2$.

Case 2. The number of updated leaf pages of primary B^+-tree is

$$DNP_P(i,t,i-1,s) = \begin{cases} Yao(\overline{fan}_{i,t}, LP_P, D_n) & , \quad XNI \leq P \\ \overline{fan}_{i,t} * \left(npu + \left[1 - \frac{npu}{np}\right]\right) & , \quad else \end{cases}$$

where $npu = Yao((1 + \sum_{t=2}^{i-2} nc_t), np, \sum_{k=1}^{n} nc_k)$

The number of 4-tuples auxiliary records being updated is

$DNR_A(i,t,i-1,s) =$

$$2 + \sum_{k=2}^{i-2} \sum_{m=1}^{nc_k} \overline{k}_{k,m}^{i-1} * \left(1 - val(k,m) * \frac{\left(\frac{Ref_{out_i-1}(k,m,n,\overline{fan}_{i-1,s})}{\overline{k}_{k,m}^{i-1}} \right)}{\left(\frac{RefVal_{A_n}(k,m,\overline{fan}_{i-1,s})}{\overline{k}_{k,m}^{i-1}} \right)} \right)$$

The second term of the sum represents the average number of tuples associated with the ancestors of $O_{i-1,s}$ that contains pointers pointing to the primary records being updated. This value is maximum when val(k,m) = 0 and is minimum when val(k,m) = 1.

Therefore, the number of pages of the auxiliary B^+-tree to be updated is

$$DNP_A(i,t,i-1,s) = \begin{cases} Yao(DNR_A(i,t,i-1,s), LP_A, \sum_{k=2}^{n} Nh_k) & , \quad \overline{AUX} \leq P \\ DNR_A(i,t,i-1,s) & , \quad else \end{cases}$$

The total cost for the case 2 is

$$C_{def}(i,t,i-1,s) = 2 * DNP_P(i,t,i-1,s)$$

$$+VisBtree\left(DNR_A(i,t,i-1,s), \sum_{k=2}^{n} Nh_k, h_A\right) + 2 * DNP_A(i,t,i-1,s)$$

The probability that case 1 occur is

$$P_1 = \frac{N_{i,t} - Def_{A_n}(i,t)}{N_{i,t}}$$

By merging the two cases, the update cost is

$$C(delete)_{i,t,i-1,s} = C_{indef}(i,t,i-1,s) * P_1 + C_{def}(i,t,i-1,s) * (1 - P_1)$$

Appendix C

Algorithm constructing a minimum auxiliary set of JIs

Input: A set of classes $C_0, ..., C_n$ and a set of target JI nodes in the path $C_0.A_1.A_2...A_n$

Output: A minimum set of auxiliary JIs nodes.

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

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

1b. For each JI in the set s in S, find its immediate auxiliary nodes.

1c. If its immediate auxiliary nodes consists of l sets, $a_1, ..., a_l$ make l copies

of s, and add each of a_i ($1< i < l$) to a copy, which forms l new sets.

1d. This process repeats until no new immediate auxiliary node can be found. The result is a set of auxiliary node sets which are used sets which are used for generating the set of target nodes.

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

3. From the retained sets obtained in step 2, calculate the number of join operations required for updating each set and select the one which requires the minimum number of join operations.

Appendix D

Estimation on the number of possible combinations

(With reference to the algorithm in previous section)

Consider step 1:

If we have a target node JI(i,j) at level n (n=j-i), the set of immediate auxiliary nodes will be:

$$JI(i,j) = JI(i,i{+}1)JI(i{+}1,j){+}JI(i,i{+}2)JI(i{+}2,j){+}...{+}JI(i,j{-}1)JI(j{-}1,j)$$

Let the searching space of a JI node be a_n where n is the level of JI node. The searching space becomes:

$$a_n = a_1 a_{n-1} + a_2 a_{n-2} + ... + a_i a_{i+1} + ... + a_{n-1} a_1$$

Observe that the right-hand side of this equation is simply the coefficient of x^n in the product g(x)g(x) $= (0 + a_1 x + ... + a_n x^n + ...)^2$

Using the power series summation method,

$$g(x) - x = \sum_{n=2}^{x} a_n x^n = \sum_{n=2}^{x} (a_1 a_{n-1} + a_2 a_{n-2} + ... + a_i a_{i+1} + ... + a_{n-1} a_1) x^n \quad = \quad (g(x))^2$$

To solve the equation :

$$\begin{aligned} g(x) - x &= (g(x))^2 \\ (g(x))^2 - g(x) + x &= 0 \\ g(x) &= \frac{1}{2}(1 \pm \sqrt{1-4x}) \end{aligned}$$

As $a_0 = 0 \Rightarrow g(0) = 0$

Therefore, $g(x) = \frac{1}{2}(1 - \sqrt{1-4x})$

Consider $(1+y)^{\frac{1}{2}} = \sqrt{1-4x}$, where y = -4x,

$$(1+y)^{\frac{1}{2}} = \sum_{k=1}^{\infty} \binom{\frac{1}{2}}{k} y^k$$

where

$$\begin{aligned} \binom{\frac{1}{2}}{k} &= \frac{\frac{1}{2}(\frac{1}{2}-1)(\frac{1}{2}-2)...(\frac{1}{2}-(k-1))}{k!} \\ &= \frac{\frac{1}{2}(-\frac{1}{2})(-\frac{3}{2})...(-\frac{1}{2}(2k-3))}{k!} \\ &= \frac{(-1)^{k-1}(\frac{1}{2})^k(1)(3)...(2k-3)}{k!} \end{aligned}$$

The coefficient of x^n in $\sqrt{1-4x}$ is

$$\begin{aligned} \binom{\frac{1}{2}}{n}(-4)^n &= \frac{(-1)^{n-1}(\frac{1}{2})^n(1)(3)...(2n-3)}{n!}(-4)^n \\ &= \frac{(-1)(1)(3)(5)...(2n-3)}{n!}2^n \\ &= \frac{(-1)(1)(3)(5)...(2n-3)}{(n!)(n-1)!}(n-1)!(2^n) \\ &= \frac{(-1)(1)(3)(5)...(2n-3)}{(n!)(n-1)!}(2)(4)(6)...(2n-2)(2) \\ &= \frac{(-1)(2n-2)!}{(n!)(n-1)!}(2) \\ &= -\frac{2}{n}\binom{2n-2}{n-1} \end{aligned}$$

Therefore, g(x) $= \frac{1}{2}(1 - \sqrt{1-4x})$ becomes :

$$g(x) = \sum_{k=1}^{\infty} \frac{1}{k} \begin{pmatrix} 2k-2 \\ k-1 \end{pmatrix} x^k$$

and a_n, the coefficient of x^n in g(x) is :

$$= \frac{1}{n} \begin{pmatrix} 2n-2 \\ n-1 \end{pmatrix}$$

Bibliography

[1] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an object-oriented database system: the story of O2*. Morgan Kaufmann, 1992.

[2] E. Bertino. An Indexing Technique For Object-Oriented Database. In *Proc. Int. Conf. Data Engineering*, pages 160–170, 1991.

[3] E. Bertino. Method precomputation in object-oriented database. In *Proceedings of ACM-SIGOIS and IEEE-TC-OA International Conference on Organizational Computing Systems (COCS91)*, 1991.

[4] E. Bertino and P. Foscoli. On Modeling Cost Functions for Object-Oriented Databases. Submitted for publication, Jan 1992.

[5] E. Bertino and P. Foscoli. Index Organizations for Object-Oriented Database Systems. *IEEE Trans. Knowledge and Data Engineering*, pages 193–209, 1995.

[6] E. Bertino and C. Guglielmina. Optimization of Object-Oriented Queries Using Path Indices. In *Proceedings of International IEEE Workshop on Research Issues on Data Engineering: Transaction and Query Processiing (RIDE-TQP)*, pages 140–149, 1992.

[7] E. Bertino and W. Kim. Indexing Techniques For Queries On Nested Objects. *IEEE Trans. Knowledge and Data Engineering*, pages 196–214, 1989.

[8] E. Bertino and L. Martino. Object-oriented database management systems: concepts and issues. *Computer (IEEE Computer Society)*, pages 33–47, 1991.

[9] E. Bertino and A. Quarati. An approach to support method invocations in object-oriented queries. In *Proceedings of International IEEE Workshop on Research Issues on Data Engineering: Transaction and Query Processiing (RIDE-TQP)*, pages 163–168, 1992.

[10] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, and H. Williams. The GemStone data management system. In W. Kim and F. H. Lockovsky, editors, *object-oriented concepts, Databases, and Applications*, pages 283–308. Addison-Wesley, 1989.

[11] R.G.G. Cattel. *Object Data Management.* Addison-Wesley Publishing Company, 1991.

[12] Sudarshan S. Chawathe, Ming-Syan Chen, and Philip S. Yu. On Index Selection Schemes for Nested Object Hierarchies. In *Proceedings International Conference on Very Large Database*, pages 331–341, 1994.

[13] S. Choenni, E. Bertino, and H. M. Blanken. On the selection off optimal index configuration in OO databases. In *Proc. Int. Conf. Data Engineering*, pages 526–537, 1994.

[14] K. A. Hua and C. Tripathy. Object skeletions: an efficient navigation structure for object-oriented database systems. In *Proc. Int. Conf. Data Engineering*, pages 508–517, 1994.

[15] A. Jhingran. Precomputation in a complex object environment. In *Proc. Int. Conf. Data Engineering*, 1991.

[16] A. Kemper and G. Moerkotte. Access Support in Object Bases. In *Proceedings ACM-SIGMOD Conference on the Management of Data*, pages 364–374, 1990.

[17] A. Kemper and G. Moerkotte. Advanced Query Processing In Object Bases Using Access Support Relations. In *Proceedings International Conference on Very Large Database*, pages 290–301, 1990.

[18] Michael Kifer, Won Kim, and Yehoshua Sagiv. Querying Object-Oriented Databases. In *Proceedings ACM-SIGMOD Conference on the Management of Data*, pages 393–402, 1992.

[19] Christoph Kilger and Guido Moerkott. Indexing Multiple Sets. In *Proceeding of the 20th VLDB Conference Santiago, Chile*, pages 180–191, 1994.

[20] K. C. Kim, W. Kim, and A. Dale. Indexing Techniques For Object-Oriented Database. In W. Kim and F. H. Lockovsky, editors, *object-oriented concepts, Databases, and Applications*, pages 371–394. Addison-Wesley, 1989.

[21] W. Kim, E. Bertino, and J. Garza. Composite object revisited. In *Proceedings ACM-SIGMOD Conference on the Management of Data*, pages 337–347, 1989.

[22] H. F. Korth and A. Silberschatz. *Database System Concepts.* McGRAW Hill, international editions edition, 1991.

[23] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[24] C. C. Low, H. Lu, B. C. Ooi, and J. Han. Efficient Access Methods in Deductive and Object-Oriented Databases. In *Proceedings International Conference on Deductive and Object-Oriented Databases*, pages 68–84, 1991.

[25] C. C. Low, B. C. Ooi, and H. Lu. H-trees: A Dynamic Associative Search Index For OODB. In *Proceedings ACM-SIGMOD Conference on the Management of Data*, pages 134–143, 1992.

[26] D. Maier and J. Stein. Indexing in an object-oriented DBMS. In *Proceedings IEEE International Workshop on Object-Oriented Database Systems*, pages 171–182, 1986.

[27] Objectivity. *Objectivity database system overview.* Objectivity, Inc., Menlo Park, CA, 1990.

[28] Ontos. *Ontos reference manual.* Ontos, Inc., Burlington, MA, 1993.

[29] Sridhar Ramaswamy and Paris C. Kanellakis. OODB Indexing by Class-Division. In *Proceedings ACM-SIGMOD Conference on the Management of Data*, pages 139–150, 1995.

[30] B. Sreenath and S. Seshadri. The hcC-tree: An Efficient Index Structure For Object Oriented Database. In *Proceeding of the 20th VLDB Conference Santiago, Chile*, pages 203–213, 1994.

[31] P. Valduriez. Join Indices. *ACM Trans. Database Systems*, pages 218–246, 1987.

[32] Versant. *Versant technical overview.* Versant Object Technologies, Inc., Menlo Park, CA, 1990.

[33] Z. Xie and J. Han. Join Index Hierarchies for Supporting Efficient Navigations in Object-Oriented Databases. In *Proceeding of the 20th VLDB Conference Santiago, Chile*, pages 522–533, 1994.

[34] S. B. Yao. Approximating Block Accesses In Database Organizations. *Communications of the ACM*, pages 260–261, April 1977.

CUHK Libraries
003510960