INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6” x 9” black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700  800/521-0600
INDEXING TECHNIQUES FOR NONTRADITIONAL DATABASE SYSTEMS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By

Wang-Chien Lee, B.S., M.S.

***

The Ohio State University

1996

Dissertation Committee:

Dr. Douglas S. Kerr
Dr. Dik Lun Lee
Dr. Renee J. Miller

Approved by

Co-Adviser

Department of Computer
and Information Science
This is dedicated to my parents.
ACKNOWLEDGEMENTS

This dissertation would have been impossible without the encouragement and support of my advisor, Dr. Dik Lun Lee. His insightful instructions and assistance led me to enter the vigorous fields of database research. In addition to the research, he also gave me a warm friendship, which I treasure greatly. Thanks also to the members of the dissertation committee, Dr. Douglas S. Kerr and Dr. Renee J. Miller, for their valuable comments.

Thanks to my fellows in the so-called 'study club'. I value each moment we had been together — whether it’s a football game at the Shoe House, a concert in the Ohio Theater, a dinner for lunar New Year, or a surprising birthday party. They are the finest. Special thanks to Tso-yu, Jondee, Ho-Hsin and others who I couldn’t enumerate, for appearing in my journey with Ohio State University and being my best friends.

Finally, I’d like to thank my parents for always trusting me and encouraging me as I chased my dreams. And thanks to my special one, Lan-Hsin, for everything. I am lucky to have such a company in my life.
VITA

May 6, 1963 ....................................................... Born - Hua-Lien, Taiwan.

1985 .............................................................. B.S., Information Science,
Chiao-Tung University, Taiwan.

1989 .............................................................. M.S., Computer Science,
Indiana University.

1989-present ................................................... Graduate Teaching Associate,
The Ohio State University.

Publications

Research Publications


Fields of Study

Major Field: Computer and Information Science

Studies in:

- Database Systems
  - Prof. Dik Lun Lee

- Computer Architecture
  - Prof. Ming T. Liu

- Distributed Computing
  - Prof. Mukesh Singhal
TABLE OF CONTENTS

DEDICATION .......................................................... ii
ACKNOWLEDGEMENTS ........................................ iii
VITA ........................................................................ iv
LIST OF TABLES .................................................. xi
LIST OF FIGURES .................................................. xiii

CHAPTER

I Introduction .......................................................... 1
  1.1 Overview ....................................................... 1
    1.1.1 Object-Oriented Databases ......................... 1
    1.1.2 Mobile Computing ...................................... 2
  1.2 Scope and Contribution ..................................... 3
  1.3 Outline of the Thesis ....................................... 5

II Nested Query Processing for Object-Oriented Databases ....... 8
  2.1 Introduction ................................................... 8
  2.2 The Basic Object Data Model ............................. 8
  2.3 Nested Object Queries ..................................... 11
  2.4 Nested Object Traversals ................................. 13
  2.5 Related Work ................................................ 16
    2.5.1 Indexing Techniques ................................. 17
    2.5.2 Signature File Techniques ......................... 21
III  Signature File Methods for Nested Object Query Processing  ............. 23
  3.1  Introduction .................................................................................... 23
  3.2  Signature File Techniques ............................................................ 24
    3.2.1  Tree Signature Scheme ......................................................... 26
    3.2.2  Path Signature Scheme ........................................................ 29
  3.3  Query Processing and Signature Maintenance .................................. 32
    3.3.1  Retrieval ................................................................................. 33
    3.3.2  Update ..................................................................................... 35
  3.4  Cost Models .................................................................................... 38
    3.4.1  Tree Signature ......................................................................... 39
    3.4.2  Path Signature ......................................................................... 40
  3.5  Summary ........................................................................................ 42

IV  Index Methods for Nested Object Query Processing .......................... 44
  4.1  Introduction .................................................................................... 44
  4.2  Path Dictionary Index ....................................................................... 45
    4.2.1  Path Dictionary ........................................................................ 46
    4.2.2  Attribute Index ........................................................................ 47
    4.2.3  Identity Index ........................................................................... 48
    4.2.4  Design Considerations ............................................................ 49
    4.2.5  s-expression Scheme ............................................................. 50
  4.3  Implementation of the s-expression Scheme ..................................... 52
  4.4  Retrieval and Update with Path Dictionary Index ............................ 56
    4.4.1  Retrieval Operations .............................................................. 56
    4.4.2  Update Operations ................................................................... 61
  4.5  Storage Cost and Performance Evaluation ...................................... 63
    4.5.1  Storage Overhead .................................................................... 66
    4.5.2  Retrieval Cost .......................................................................... 73
    4.5.3  Range Query Cost ................................................................... 78
    4.5.4  Update Cost ............................................................................. 81
    4.5.5  Summary of Formulae for Path Index and Path Dictionary
          Organizations ............................................................................. 86
  4.6  Direct Links Organization .............................................................. 87
    4.6.1  Definition of Direct Links ....................................................... 88
    4.6.2  Direct Links Indexes ............................................................... 89
    4.6.3  Analytical Cost Model ............................................................ 94
4.7 Summary ......................................................... 103

V Information Broadcast in Mobile Computing Environments .......... 105
  5.1 Introduction .................................................. 105
  5.2 Mobile Computing System Architecture .................. 105
  5.3 Power Conservation and Information Broadcast .......... 107
  5.4 Signature Caching Policies .............................. 111
  5.5 Related Work ............................................... 113
    5.5.1 Hashing Techniques .................................. 113
    5.5.2 Indexing Techniques ................................. 115
    5.5.3 Caching Techniques ................................. 115

VI Signature Techniques for Wireless Information Broadcast ........ 118
  6.1 Introduction .................................................. 118
  6.2 Information Broadcast Using the Signature Techniques ...... 119
    6.2.1 Simple Signature Scheme .......................... 121
    6.2.2 Integrated Signature Scheme ...................... 123
    6.2.3 Multi-level Signature Scheme ...................... 126
    6.2.4 Performance Analysis ............................. 129
    6.2.5 Adaptation of Signature Sizes ..................... 143
  6.3 Summary ......................................................... 147

VII Signature Caching Techniques for Wireless Information Broadcast 149
  7.1 Introduction .................................................. 149
  7.2 Signature Caching Policies ................................ 152
    7.2.1 Two-level Signature Scheme ...................... 152
    7.2.2 Bit Tags and Active Refresh (BA) .................. 154
    7.2.3 Version Numbers and Active Refresh (VA) ....... 156
    7.2.4 Bit Tag and Passive Refresh (BP) .................. 158
    7.2.5 Version Numbers and Passive Refresh (VP) ...... 160
  7.3 Comparison of Signature Caching Policies ................ 162
  7.4 Limiting The Cache Size .................................. 169
    7.4.1 Cache Replacement Policies ....................... 169
    7.4.2 Performance Evaluation of the Cache Replacement Policy 172
  7.5 Summary ......................................................... 175
VIII Summary and Future Directions .............................. 177

8.1 Summary ................................................................. 177
  8.1.1 Object-Oriented Database Systems ............... 177
  8.1.2 Mobile Computing ........................................ 180
8.2 Contributions ....................................................... 183
8.3 Future Directions ................................................. 185

APPENDICES

A Proof of Lemmas ................................................... 188
LIST OF TABLES

TABLE PAGE

1 Parameters and symbols for cost models of tree and path signature schemes. 38
2 Parameters and symbols for cost models of path index and path dictionary index. 65
3 Parameter values for the cost models of path index and path dictionary index. 66
4 Summary of formulae for path index and path dictionary organizations. 86
5 Parameters and symbols for cost models of direct link indexes. 95
6 Parameter values for the cost models of direct link indexes. 95
7 Summary of formulae for direct link organizations. 102
8 Parameters and symbols for cost models of simple, integrated and multi-level signature schemes. 131
9 Parameter values of the cost models for simple, integrated and multi-level signature schemes. 138
10 Signature size, access time w.r.t. minimal tune time. 141
11 Parameters and symbols for cost models of signature caching policies. 153
12 Parameters values for the cost models of signature caching policies. 162
13 Actions for cache replacement. ............................ 171
# List of Figures

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Aggregation hierarchy.</td>
</tr>
<tr>
<td>2</td>
<td>Inheritance hierarchy.</td>
</tr>
<tr>
<td>3</td>
<td>Signature generation and comparison.</td>
</tr>
<tr>
<td>4</td>
<td>An aggregation hierarchy indexed with tree signatures.</td>
</tr>
<tr>
<td>5</td>
<td>Tree signature scheme. (a) Format of a signature file; (b) creation of an object signature.</td>
</tr>
<tr>
<td>6</td>
<td>An aggregation hierarchy indexed with path signatures.</td>
</tr>
<tr>
<td>7</td>
<td>Path signature scheme. (a) Format of a signature file; (b) creation of an object signature.</td>
</tr>
<tr>
<td>8</td>
<td>Path dictionary index.</td>
</tr>
<tr>
<td>9</td>
<td>(a) A database instance, (b) Path information.</td>
</tr>
<tr>
<td>10</td>
<td>Examples of the s-expression scheme.</td>
</tr>
<tr>
<td>11</td>
<td>Data structure of an s-expression.</td>
</tr>
<tr>
<td>12</td>
<td>Path dictionary index. (a) Structure of the path dictionary index; (b) leaf node record of the identity index; (c) nonleaf node of the identity index; (d) leaf node record of an attribute index; (e) nonleaf node of an attribute index.</td>
</tr>
</tbody>
</table>
13 Storage overhead. ................................................................. 71
14 Storage overhead for multiple attribute indexes \((K = 3)\). ............. 72
15 Retrieval cost for a single TP query. ........................................ 76
16 General retrieval cost for mixed TP queries. ............................... 77
17 Range retrieval cost \((k_1 = k_2 = k_3 = q = 3)\). .............................. 80
18 Update cost. ........................................................................... 84
19 Direct links index. ................................................................. 89
20 Structures of attribute indexes. ................................................ 93
21 Storage overhead. ................................................................... 98
22 Retrieval cost. ....................................................................... 101
23 Wireless mobile system architecture. ......................................... 106
24 Information stream. .............................................................. 109
25 Non-interleaved signatures. ..................................................... 120
26 Interleaving signatures. .......................................................... 120
27 Integrated signature scheme. .................................................. 124
28 Multi-level signature scheme. ................................................. 126
29 Access time vs signature size. ............................................... 140
30 Tune-in time vs signature size. .............................................. 140
31 Tune-in time vs access time \((l = 3; s_i = 280)\). .............................. 142
32 Tune-in time vs access time ($l = 1; s_i = 400$) ................................. 143
33 Tune-in time vs integrated signature share ........................................... 146
34 Minimal tune-in time vs access time .................................................... 147
35 Tune-in time vs query number ............................................................... 163
36 Tune-in time vs signature size ............................................................... 165
37 Tune-in time vs access time ................................................................. 166
38 Tune-in time vs group size ................................................................. 168
39 Tune-in time vs update percentage ...................................................... 169
40 Hits vs query number ........................................................................... 173
41 Hits vs cache size .................................................................................. 174
CHAPTER I

Introduction

1.1 Overview

Object-orientation and mobility are two of the most influential ideas in the development of database systems introduced in recent years. The former is aimed at extending database facilities to support complex applications, while the latter is focused on allowing database users to access information without geographical limitation. These two nontraditional aspects of database systems are two of the most exciting database research areas in recent years.

1.1.1 Object-Oriented Databases

Object-oriented database systems (OODBSs) have been one of the most prominent areas of database research in the last decade. Many experimental prototypes [16,32,43] and commercial systems [20,29,33] have been introduced. In addition to facilitating the design and engineering of traditional database applications, OODBSs provide data modeling mechanisms to support flexible data types, useful relationships (e.g., aggregation/association and specialization/generalization relationships) and allow users to define, query, and update nested entities. This functionality meets the needs of new
database applications such as CAD/CAM, CASE, office automation, multimedia systems, and geographic information systems.

The need to reduce the cost of developing, operating, and supporting the above mentioned applications prompted the rapid development of OODBSs. However, implementation and performance issues play an important role in the success of OODBSs. Efficient techniques for query processing and indexing are critical to these issues [1,9,25,27,28,30]. Although indexing techniques have been proposed to support query processing in OODBSs [10,11,31,42], they in general introduce large storage overhead and maintenance cost. In the first part of this thesis, we investigate the problems of nested query processing in OODBSs and propose new indexing organizations and the associated query processing methods.

1.1.2 Mobile Computing

Owing to the marriage of personal computing and wireless networking technologies, mobile computing becomes a rapidly exploding area in the past few years. Industry, government, and the academia are all anticipating the up-coming personal communication and computing era [15,19]. Various commercial and experimental mobile computers\(^1\), e.g., Palmtops and personal communication systems (PCSs), have appeared recently [7,52]. It is envisioned that mobile computers will be as popular as walkmans and portable TVs in the near future and promise to revolutionize the information service market. In a wireless communication environment, users with mobile

\(^1\)We use mobile computers, mobile hosts, and mobile clients interchangeably throughout the thesis.
computers are able to access a large number of databases and information services and communicate with other users via wireless communication. For example, while on the road, mobile users may receive traffic and weather information on the air \cite{47}; investors may access the most current information about stock prices; in a shopping mall, customers may query about the stores, products, and sale information \cite{4}.

To meet the market needs, ubiquitous services, such as fax-oriented messengers, nomadic conferencing and computing, mail-triggered applications and information broadcasting, will emerge as some of the most important research topics in the next decade \cite{21}. In the second part of this thesis, we consider the indexing problems with information broadcast services for mobile users equipped with battery powered mobile computers.

1.2 Scope and Contribution

For a database system, the performance of query processing is critical to its success. Learning from the development history of traditional database systems, we know that indexing techniques play an important role in the efficiency of query processing. Therefore, we can expect that indexing techniques will play an equally important role in query processing on nontraditional database systems such as OODBSs and mobile computing systems. The problems investigated in this thesis can be stated as follows.

1. How can indexing techniques be applied to nested object query processing in OODBSs.
2. How can indexing techniques be applied to information broadcast services in mobile computing environments?

3. What other techniques can be used to optimize the performance of OODBSs and mobile computing systems?

To address the problems, we investigate the characteristics of query processing for OODBSs and mobile computing systems. Based on the characteristics of the systems, we propose different indexing organizations to improve the performance of the systems. Furthermore, we study other optimization methods to improve the performance of the systems. The main contributions of this thesis are as follows:

1. A classification of nested object queries. The classification illustrates how to process various kinds of nested object queries. In particular, we found that the indexing organizations in the literature don't support all of the queries classified. The classification contributes greatly to our finding that path information may be used for processing nested object queries.

2. Two signature file schemes, tree signature and path signature, for query processing in OODBSs. These two signature schemes are the first signature methods proposed for the OODBSs in the literature. They have much simpler file structure than the traditional tree-based inverted indexes with small storage overheads.

3. Two classes of indexing organizations, path dictionary and direct links, for query processing in OODBSs. These two indexing organizations realize the
idea of utilizing path information for indexing OODBSs. They may be flexibly extended to index on arbitrary attributes of the databases with small storage and update overhead, while significantly improving the speed of object retrieval.

4. Three signature file schemes, simple signature, integrated signature and multilevel signature, for mobile computing systems. These schemes dramatically reduce the requirement for battery power of the mobile computers, which is a critical factor for the mobility of the mobile computers. We also demonstrated that the signature schemes are natural organizations for indexing broadcast information.

5. Signature caching policies for mobile computing systems using multilevel signature schemes. Through the study of signature caching policies, we address the issues of caching in the wireless and mobile environments. The caching policies are important due to physical constraints of the mobile computers.

1.3 Outline of the Thesis

This thesis is divided into the following chapters. Chapter 2 is a detailed study on the characteristics of nested query processing for OODBSs. We first provide some background on object-oriented data model to facilitate our discussion on nested object queries. A classification of the nested queries, based on their characteristics, is presented. We then look in detail into the traditional query processing approaches for nested object queries. Finally, we review indexing and signature techniques for nested object query processing as proposed in the literature.
Chapter 3 presents the signature file techniques for nested query processing of OODBs. We introduce the concept of signature file techniques and propose tree signature scheme and path signature scheme for nested object query processing. Retrieval and update operations supported by our signature file organizations are described. Cost models for the operations of these two signature schemes are developed.

Chapter 4 presents the indexing techniques utilizing path information embedded in the databases for processing of nested object queries. We first introduce the concept of path information, path dictionary, attribute index, identity index and the $s$-expression scheme for realizing the path dictionary. Then the implementation of path dictionary index with $s$-expressions is presented in detail. The retrieval and update operations supported by the path dictionary index organizations are described. Cost models for storage overhead and retrieval and update operations are developed. A performance evaluation which compares the path dictionary approaches with the well-known path index is conducted. Finally, we present the direct link organization, which is a variation of the path dictionary. Structure of the direct links and its operations are described in detail. Analytical cost models and performance evaluation are also developed and conducted.

Chapter 5 discusses the problems with indexing information broadcast services. We first describe the architecture of mobile computing systems. Then we address the issues of power conservation and information broadcast in wireless and mobile environments. Related work in the literature is then surveyed.
Chapter 6 presents the signature method for query processing in mobile computing systems. We first list the advantages of using signature and caching techniques for information broadcast services. Three signature schemes for indexing the information broadcast services are described in depth. Cost models for these three schemes are developed and used to evaluate their performance in terms of tune-in time and access time. Finally, the issues of optimizing signature sizes for the multi-level signature scheme are addressed.

Chapter 7 further explores the problems of signature caching in information broadcast services. We first discuss factors to be considered for signature caching policies. Based on these factors, four signature caching policies for the multi-level signature scheme are proposed and analyzed. Performance of the four policies are compared by varying a number of parameters. Finally, we consider cache replacement policies for mobile computers with small cache size. Two policies were discussed and compared by simulation.

Finally, Chapter 8 summarizes the scope and contribution of this thesis and depicts a number of ideas for future research that will expand the work presented in this thesis.
CHAPTER II

Nested Query Processing for Object-Oriented Databases

2.1 Introduction

This chapter lays the background for our study on nested object query processing. We first discuss the basic concepts of the object data model within the scope of this thesis. Then we classify the nested object queries into different categories and provide examples for them. Finally, we discuss the traditional traversal approaches for evaluation of nested object queries and related work for indexing OODBs.

2.2 The Basic Object Data Model

The object-oriented data model is based on the following basic concepts. In the data model, a real-world entity is represented as an object, which consists of methods and attributes. While methods, which are procedures and functions associated with the object, define the reactions of the object in response to messages from other objects, attributes represent state of the object. Objects sharing the same methods and attributes are grouped into classes. The concept of class allows OODBs to model complex data more precisely and conveniently than the relational data model. A
class may consist of simple attributes (e.g., of domain integer or string) and complex attributes with user-defined classes as their domains. Since a class $C$ may have a complex attribute with domain $C'$, an aggregation relationship can be established between $C$ and $C'$. Using arrows connecting classes to represent aggregation relationship, a directed graph, called the aggregation hierarchy, may be built to show the nested structure of the classes.

Figure 1 is an example of an aggregation hierarchy, which consists of four classes, Person, Vehicle, Person_Name, and Company. The class Person has three simple attributes, SSN, Residence and Age, and two complex attributes, Owns and Name. The domain classes of the attributes Owns and Name are Vehicle and Person_Name, respectively. The class Vehicle is defined by three simple attributes, Id, Color, and Model, and a complex attribute Manufacturer, which has Company as its domain. Company and Person_Name each consists of two simple attributes.
Another feature of OODBS is the specialization/generalization relationships between classes, called inheritance relationships. An inheritance relationship organizes classes into an inheritance hierarchy. Inheritance allows a class $C$ to be defined as a specialization of another class $C'$. $C$ is called the subclass of $C'$ and $C'$ is a superclass of $C$. A subclass inherits attributes and methods from its superclasses. A subclass can have more attributes and methods than its superclass.

Since similar objects are grouped into a hierarchy of classes, whether to create a single index for the whole hierarchy of classes or to build an index for each of the classes is an interesting indexing problem.

Figure 2: Inheritance hierarchy.

Figure 2 is an example of inheritance hierarchy among the class Vehicle and its subclasses Automobile, DomesticAuto, ForeignAuto, and Truck. In addition to the
attributes inherited from the superclass Vehicle, objects in class Automobile have an additional attribute Model. Therefore, the objects in class DomesticAuto have attributes Id, Color, Manufacturer, Model, and Country.

Every object in an OODBS is identified by an object identifier (OID). The OID of an object may be stored as an attribute value of other objects. If an object $O'$ is referenced as an attribute of object $O$, $O'$ is said to be nested in $O$ and $O$ is referred to as the parent object of $O'$. The parent objects of an object $O$ are considered as an ancestor object of $O$. Moreover, the ancestor objects of an object $O$ are also ancestor objects of the objects nested in $O$. Objects are nested according to the aggregation hierarchy. Thus, the classes in the aggregation hierarchy are similarly named based on their nested relationship.

2.3 Nested Object Queries

OODBSs support queries involving nested objects. These queries are called nested queries. There are many kinds of nested queries. However, an access method doesn't necessarily support all of them. Even with the same access method, different kinds of queries may be evaluated differently. To facilitate our discussion, we define target classes as the classes from which objects are retrieved and predicate classes as the classes involved in the predicates of the query. We classify nested queries by the following factors:

1. Relative positions of the target and predicate classes on the aggregation hierarchy:
• TP: The target class is an ancestor class of the predicate classes.

• PT: The target class is a nested class of the predicate classes.

• MX: The target class is an ancestor class of some predicate class and a nested class of some predicate class.

2. The complexity of the predicates:

• Simple: The predicate is specified on a simple attribute. Based on the operators used in the predicates, this class of nested queries is further divided as follows.
  
  - Equality: =.
  
  - Range: >, ≥, <, ≤, *, and ?\(^1\).
  
  - Inequality: ≠.

• Complex: The predicate is specified on a complex attribute. Depending on whether or not an OID is specified in the predicate class, this class can be further divided into:

  - Exist: An OID is specified in the predicate.

  - Nonexist: No OID is specified in the predicate.

In the following, we give examples for each type of the nested queries. We will use some of these examples to illustrate the query processing strategies described in

\(^1\) * and ? are partial string match operators. * matches an arbitrary number of characters, while ? matches one character.
the following chapters. Note that Company[i] and Person[i] denote the OIDs of the ith object in class Company and Person, respectively.

Q1: Retrieve persons who own cars made by "GM". (TP-Simple-Equality)

Q2: Retrieve persons who own cars made by "G*". (TP-Simple-Range)

Q3: Retrieve persons who own cars not made by "GM". (TP-Simple-Inequality)

Q4: Retrieve manufacturers of the cars owned by persons at the age of 50. (PT-Simple-Equality)

Q5: Retrieve manufacturers of the cars owned by persons older than 50. (PT-Simple-Range)

Q6: Retrieve manufacturers of the cars owned by persons not at the age of 50. (PT-Simple-Inequality)

Q7: Retrieve persons who have a car made by Company[1]. (TP-Complex-Exist)

Q8: Retrieve persons who don't have a car. (TP-Complex-Nonexist)

Q9: Retrieve vehicles which are owned by Person[1]. (PT-Complex-Exist)

Q10: Retrieve vehicles which are not owned by any person. (PT-Complex-Nonexist)

2.4 Nested Object Traversals

There are three basic approaches to evaluating a nested query: top-down, bottom-up and mixed evaluations. The top-down approach traverses the objects starting from
an ancestor class to a nested class. Since the OID in a parent object leads directly to a child object, this approach is also called a forward traversal approach. On the other hand, the bottom-up method, also known as backward traversal, traverses up the aggregation hierarchy. A child object, in general, does not carry the OID of (or an inverse reference to) its parent object. Therefore, in order to identify the parent object(s) of an object, we have to compare the child object's OID against the corresponding complex attribute in the parent class. This is similar to a relational join when we have more than one child object to start with. Mixed evaluation is a combination of the top-down and bottom-up approaches, which is often required for complex queries. Note that when every reference from an object \( O \) to another object \( O' \) (e.g., \( \text{owns} \)) is accompanied with an inverse reference from \( O' \) to \( O \) (e.g., \( \text{Owned_by} \)), the aggregation hierarchy becomes bi-directional, resulting in no difference between the top-down and the bottom-up approaches. In this thesis, however, we assume there are no inverse references.

Let’s consider the above query examples. To answer Q1 in the top-down approach, the system has to retrieve all of the objects in class Person, then retrieve the Vehicle objects of the Person objects and their nested Company objects to check the manufacturers' names. Finally, those persons who own GM cars are returned. In the bottom-up approach, the objects in class Company are retrieved to examine if their names are GM. The OID’s of the GM companies are maintained in a set \( S \). Then, the vehicle objects in class Vehicle are examined to identify those vehicles made by the companies in \( S \). The qualified vehicle objects are collected in a set \( S' \). Finally,
the Person objects are retrieved to find out if their cars are one of the vehicles in $S'$. The other TP queries, such as Q2, Q3, Q7 and Q8, can be evaluated similarly by using either top-down or bottom-up traversal approaches.

For PT queries, which retrieve nested attributes of some specific collection of objects, the top-down approach is effective. Take Q4 as an example, the objects in Person are first retrieved to examine their ages. Those objects with Age of 50 are then traversed along the path of Person.Vehicle.Company to retrieve the names of the automobile makers. The bottom-up approach, in contrast to the top-down approach, is cumbersome in this query. It requires all of the objects in class Company to join to the objects in class Vehicle, and the result is further joined to the Person class. Then, the names of the auto makers corresponding to the 50-year-old auto owners are returned.

The other PT queries, except for Q10, may be evaluated in a manner similar to Q4. To answer Q10, we have to retrieve person objects and collect in a set $S$ the vehicle objects which have owners. Then, the set $S$ is subtracted from the Vehicle class to return the vehicle objects which have no owners.

The performance of the top-down and the bottom-up approaches is strongly dependent on the distribution of objects located in the classes to be traversed. No matter where the predicate classes are located, the bottom-up approach will always scan through all objects in the classes on the path. However, the number of objects fetched by the top-down approach is at most the number of objects in the top class times the number of classes on the path.
As a result, it is intuitive to conclude that the top-down evaluation is more appropriate than the bottom-up method when the ancestor classes have fewer objects than the other classes on the paths or when the number of path traversals is reduced due to the predicate evaluations on the top classes. On the other hand, the bottom-up evaluation is more suitable when the ratio of object sharing from ancestor classes to their nested classes is high. In other words, the bottom-up approach prevails when the numbers of objects in ancestor classes are much larger than that of their nested classes.

Generally speaking, the top-down approach has an advantage over the bottom-up approach for queries which have predicate classes located near the top of the paths to be traversed. If the number of qualified objects in these predicate classes is small, the number of forward traversals will be small. For example, if there are 10 persons who are 50 years old in the class Person, at most 10 Vehicle objects and 10 Company objects will be fetched. For this kind of queries, the top-down method is a better choice than the bottom-up method. Another disadvantage of the bottom-up approach is that it requires a lot of internal memory due to the breadth-first style of the join operations. On the other hand, the top-down approach may choose a depth-first style of forward traversal which doesn't require much internal memory.

2.5 Related Work

A relational database consists of a group of individual relations. The relations are related through primitive key values. The join operation is used to connect these
relations. In object-oriented databases, objects of various classes are related by object identifiers, which leads to the nested structure of objects. Traversal through the bridges built upon OIDs is a natural way of evaluating OODBs queries. Therefore, nested queries implies traversal of objects along the path between the target class and the nested attributes.

From our discussion on traversal methods, we can see that a significant part of the query processing cost is spent on accessing intermediate objects between the target class and the predicate classes. Techniques based on indexing or signature file methods have been proposed to expedite the processing of queries. According to our observation, the essence of these techniques is to reduce physical traversals of intermediate objects between the target class and the predicate classes.

2.5.1 Indexing Techniques

The idea behind indexing techniques for nested query processing is to map a value of certain attribute to some ancestor objects which directly or indirectly own the attribute value. The indexing mechanisms implicitly create a direct reverse link from a nested attribute to an ancestor class. As a result, the goal of bypassing the intermediate objects is achieved by scanning indexes. Indexing techniques are effective and will be efficient as long as the overhead they introduce is smaller than the saving gained from avoiding intermediate object traversal. Unfortunately, most indexing techniques require costly storage overhead and expensive index maintenance. Therefore, they can't be applied on too many attributes. Only some frequently queried target classes and predicate attributes can be chosen to create indexes.
Indexing Aggregation Hierarchy

Multiple Index, proposed by Maier and Stein [42], is the first of the indexing techniques for OODBSs. It creates an index for each edge on the path from a nested attribute to the target class. It is like creating a reverse link for each edge along the path. To answer a query involving the indexed attribute and target class, index scans may be used to replace physical access to intermediate objects for backward traversals from the nested attribute to the objects in target class. Although several indexes are created for a given path and thus many index scans are necessary for a query evaluation, this organization is flexible for creating indexes sharing a path without introducing much duplicated overhead.

Nested Index and Path Index were proposed by Bertino and Kim [11]. These two indexes map a specific nested attribute to the target class and to the classes located along the given path, respectively. Like multiple index, they separately create implicit reverse links from the nested attribute to the target class and the classes appearing on the path. Only one index scan is needed to reach the target classes from the nested attribute. Although both techniques are very effective, they require high storage cost and expensive update maintenance. Thus, they are very expensive when many attributes are indexed. Further, the nested index requires system-supported reverse links among objects in the path to efficiently update the index [11].

These indexing techniques cannot support all of the nested queries we classified. PT queries implicitly suggest a forward traversal to the nested attributes. Thus, they cannot benefit from the reverse links built by the indexes. Support of TP-
Simple-Inequality queries is problematic, since inequality cannot be easily supported by indexing techniques. We can create indexes for TP-Complex-Exist queries using OIDs as the key. However, the applicability of these indexes is limited, because the only meaningful operator for the kind of indexes is equality. Finally, TP-Complex-Nonexist queries are not supported by these indexing techniques either.

*Field Replication Technique* was proposed by Shekita and Carey [48]. As its name suggested, attributes of nested objects are replicated into their ancestor objects. Therefore, nested attribute values that would normally be accessed through forward traversal are replicated such that expensive traversals may be avoided. The problems with field replication are that it imposes a structure change to the original database and that its update cost is expensive. In order to improve the update performance, *inverted path* was introduced to implement reverse links along the path [48]. The idea of inverted path is similar to that of multiple index. Therefore, this organization can be used to support backward traversals and some of the forward traversals where the nested attributes are replicated.

*Join Indices* were proposed by Valduriez for improving joins in relational database systems [51]. It may be created by joining two relations, say $R$ and $S$, and project the corresponding tuple identifiers from $R$ and $S$ into a (join index) relation of 2 attributes. In order to facilitate fast access to the join index relation, two copies of the join index are usually maintained. One copy is clustered on the tuple identifiers of $R$ and the other is clustered on that of $S$. The join indices can also be used in OODBSs. A join index may be used for each direct connection of classes along a
given path. Therefore, it may be implemented as two sets of multiple indexes, which will allow both directions of traversal. The tradeoff for the bi-directional traversal is to double the storage overhead.

*Access support relations*, by Kemper and Moerkotte, are generalization of the join indices for OODBs [26]. Instead of supporting traversal (or join) of two connected classes (relations), access support relations support the traversal along a path of arbitrary length. The relations may be created by joining all of the classes along the path and projecting the object identifiers from the classes on the path. Similar to join indices, two copies of an access support relation are stored and clustered correspondingly on the OIDs of objects in the two end classes of the path. Therefore, traversals from either end class of the path to any class on the path can be supported.

**Indexing Inheritance Hierarchy**

Issues in building a single index for the classes in an inheritance hierarchy were studied by Kim, Kim and Dale in [31]. It compares the *Single-Class Index*, which maintains a conventional index for each class of the hierarchy, to the *Class-Hierarchy Index*, which maintains a single index for the whole hierarchy of classes. The conclusion is that the class-hierarchy index is superior to the single-class index as long as there are more than two classes in the inheritance hierarchy [31]. However, the study in [31] is confined to single level indexes for primitive attributes of a class without considering nested attributes.

*H-tree* is a hierarchical indexing organization proposed by Low, Ooi and Lu for supporting efficient associative search on objects based on the inheritance hierarchy.
The organization is tailored to supporting object retrieval from a single class as well as from an inheritance hierarchy of classes. The H-tree indexes of the classes are structured in accordance with the inheritance hierarchy. A $B^+$-tree is created for each class. The nested indexes are connected to their parent indexes by pointers, which associate the nodes in the nested H-trees to their parent H-trees according to the indexed values. As a result, searching for a value against an inheritance hierarchy of classes requires only a full search on the root H-tree and partial searches on the nested H-trees.

The study of Nested-Inherited Index by Bertino [8] takes both of the aggregation and inheritance relationships into account. Actually the inheritance index scheme is an extension of the path index to cover the inheritance relationship among classes. Yet another improvement of the nested-inherited index over the path index is to store the ancestor information of objects in a network of indexed auxiliary records. This mechanism facilitates the update maintenance of the nested-inherited index. However, the storage overhead involved is tremendous.

### 2.5.2 Signature File Techniques

The signature file techniques, in contrast to the indexing techniques, use abstracted information stored in signature files to avoid actual retrieval of intermediate objects located on the paths from the top class to the nested attributes. An object signature is an abstraction of the information stored in the (nested) attributes of the object. When processing a query, a query signature is formed to match with object signatures. An object signature which fails to match the query signature guarantees that
the corresponding object can be ignored. Consequently unnecessary object accesses are avoided. Only the objects passing the signature matching are traversed to verify if they really satisfy the query. The signature file techniques generally have a much lower storage overhead and a simpler structure than indexing techniques. They are particularly good for queries which require forward traversals (i.e., PT queries). Moreover, they are good for queries involving a large number of the attributes, because the signatures can index on many attributes.

*Signature replication technique*, proposed by Yong, Lee and Kim [54], generates object signatures from the direct attribute values of the objects. Instead of using only OIDs, the object signature of an object and its OID are stored as complex attributes of the parent objects. Therefore, the object signatures are first used to screen out unqualified nested objects before performing forward traversal. Like the field replication technique, this organization will change the structure of the original database. Further, it has expensive update and maintenance costs. Also, for a nested query involving a long path with predicate class at the far end, the signature replication method may save only the final step in the traversal instead of bypassing all intermediate classes in the path.
CHAPTER III

Signature File Methods for Nested Object Query Processing

3.1 Introduction

Signature file methods have been used extensively for various database applications, such as text retrieval [17], image database [36], multimedia database [45,50] and other conventional database systems [13]. The main idea behind the signature file methods is to store abstracted information about objects into the object signatures. By checking the signature of an object, we may predict whether the object will satisfy a query. Therefore, signature file techniques are especially good for avoiding unnecessary object traversals in OODBSSs.

In this chapter, we first describe the concepts of signature file techniques and the structures of two signature file schemes, namely, tree signature and path signature, for nested object query processing. Next, we present the algorithms for query processing and signature files maintenance involving update operations. Then a cost model is proposed to facilitate the analysis of the overhead cost and performance of the signature schemes.
3.2 Signature File Techniques

<table>
<thead>
<tr>
<th>object</th>
<th>John</th>
<th>123456789</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute signatures:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h(John) )</td>
<td>000 010 101 001</td>
<td></td>
</tr>
<tr>
<td>( h(123456789) )</td>
<td>001 000 110 010</td>
<td></td>
</tr>
<tr>
<td>object signature (V)</td>
<td>001 010 111 011</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Queries</th>
<th>Query Signatures</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Jack</td>
<td>010 001 000 011</td>
<td>← no match</td>
</tr>
<tr>
<td>2a) John</td>
<td>000 010 101 001</td>
<td>← true match</td>
</tr>
<tr>
<td>2b) 111223333</td>
<td>001 000 111 000</td>
<td>← false drop</td>
</tr>
</tbody>
</table>

Figure 3: Signature generation and comparison.

A signature is basically an abstraction of the information stored in the (nested) attributes of an object [46]. It is a superimposed bit string generated from the attribute values of the object being represented. Figure 3 depicts the signature generation and comparison processes of an object having two primitive attributes, name and SSN. Signatures of primitive attributes are obtained by hashing their attribute values into a bit string. An object signature is formed by superimposing the signatures of its attributes. Object signatures are stored sequentially in a signature file. A query specifying certain values to be searched for is transformed into a query signature \( S_Q \) in a similar way. The query signature is then compared to every object signature in the signature file. The figure illustrates three possible outcomes of the comparison:

1. the object and query signatures don't matches (i.e. \( S_Q \land S_i \neq S_Q \));

2. the object and query signatures match (i.e. \( S_Q \land S_i = S_Q \));
a) the object really satisfies the search criteria;

b) the signature comparison indicates a match, but the object in fact does not satisfy the search criteria.

Case 2a is called a true match and case 2b is called a false drop. In order to eliminate false drops, the object must be examined after the object signature signifies a match.

The purpose of using a signature file is to screen out most of the unqualified objects. A signature failing to match the query signature guarantees that the corresponding object can be ignored. Therefore, unnecessary object accesses are prevented. Signature files have a much lower storage overhead and a simpler file structure than inverted indexes. They are particularly good for multi-attribute retrieval when the attributes have an equal chance of being specified in the query. Since queries in an OODBS could be very flexible, inverted indexes would be very complex if all possible access paths are supported. However, as can be seen later, the signature file approach offers a much simpler solution at the expense of some retrieval speed.

In this section, we present two signature file techniques to support efficient object-oriented database query evaluation. Queries having a single class as target are used in our discussion. However, our methods can be directly applied to queries against multiple classes in an inheritance class hierarchy. Also, we assume that there is no loop in the aggregation hierarchy.
3.2.1 Tree Signature Scheme

For every class $C$ in the aggregation hierarchy, there exists a signature file $S$ such that every object $O$ in $C$ has an entry $(\text{sig}, \text{oid})$ in $S$, where $\text{sig}$ is the signature of $O$ and $\text{oid}$ is the OID of $O$. The signature of an object can be recursively defined as follows.

1. The signature of an object is generated by superimposing the signatures of all of its primitive and complex attributes;

2. The signature of a primitive attribute is obtained by hashing on the attribute values; the signature of a complex attribute is the signature of the object it references.

Figure 4: An aggregation hierarchy indexed with tree signatures.
For each class in the aggregation hierarchy, a file of signatures is created for the objects in the class. As shown in Figure 4, a signature file is associated with each class in the aggregation hierarchy. The information stored in a signature file is the superimposed information of all (nested) attributes of its associated class. Figure 5 shows the format of the signature file for class Person. We assume that there are $N$ objects in the class and use $\text{Person}[i]$ to represent the object identifier of the $i$th object in $\text{Person}$. The signature file $\text{Person.Sig}$ of class $\text{Person}$ consists of $N$ signatures, one for each object in the class. As shown in Fig. 5(b), the signature of an object in $\text{Person}$ is the superimposition of the signatures of its primitive attributes, SSN, Residence and Age, and all of the primitive attributes nested under Owns and Name.

![Figure 5: Tree signature scheme. (a) Format of a signature file; (b) creation of an object signature.](image)
Since a signature for an object $O$ is formed by superimposing the signatures from all of the nested primitive attributes of $O$, the signature of $O$ can be used directly to generate the signatures of its parent objects when the parents’ signatures are computed. A parent object $P$ of $O$ indirectly references all of the nested attributes of $O$. Thus, $P$’s signature contains all of the abstract information from $O$ and other attributes of $P$. In other words, the signature of $P$ contains more information than $O$. Because of the nature of superimposing coding, the higher level an object is located in the aggregation hierarchy, the more information its signature has to cover. Therefore, the signature will give more false drops. To alleviate this problem, we may choose to hash only the more frequently used attributes into the signatures. However, information about hashed attributes has to be maintained for query processing. In the thesis, we only consider the tree signatures created by hashing all of the nested primitive attributes.

When evaluating a query, e.g., “Retrieve all red vehicles manufactured by Fiat.”, we can take advantage of the signature files on both top-down and bottom-up evaluations. For the top-down approach, a signature for “red” $\land$ “Fiat” is generated. Then the signature is used to match with the signature file of class Vehicle. A set of matched objects is collected. The set of objects are fetched to verify if they are red. The path from the manufacturer of the vehicle to the name of the company is traversed to verify if the car is made by Fiat. The verification is necessary because of false drops and because attribute names are not presented in the signatures. Thus, a company with location = “Fiat” might have been retrieved if verification is not done.
For the bottom-up approach, a signature for “Fiat” is generated. The signature is used to match with the signature file of class Company. A set of matched objects are collected and retrieved for verification. The signatures of the qualified company objects are used to match with the signature file of class Vehicle. Finally the set of matched vehicle objects is retrieved to eliminate false drops and to verify if the color is red.

### 3.2.2 Path Signature Scheme

In the tree signature scheme, a signature for an object $O$ is created from all of the primitive values in the subtree of objects rooted at $O$. The filtering capability of the signature technique is weakened if the subtree is large. The path signature scheme creates signature files for a single path. The signature files created for the classes on a path will support the evaluation of queries involving any classes on the path and their attributes. Since the attributes covered in a path signature are limited to those on a path, the number of attributes hashed into path signatures is smaller than that hashed into tree signatures. With the same signature length, the filtering capability of the path signatures is expected to be better than that of the tree signatures. Therefore, the path signature scheme is especially useful for applications where some paths are frequently specified in queries.

Similar to the tree signature scheme, the signature of an object $O$ in the path signature scheme is generated by superimposing the signatures of $O$’s attributes. However, in the path signature scheme, the signatures of complex attributes located on the path and those not located on the path are generated in different ways. The
signatures of complex attributes located on the path are generated by superimposing the signatures of its nested attributes, while the signatures of other complex attributes are generated by using the object identifiers stored in the attributes as the keys for hashing. In other words, the complex attributes which are not located on the path are treated as primitive attributes by taking their OIDs as primitive values. Another difference between the path and tree signature schemes is that the path signature scheme uses a list of OIDs, which includes the object identifiers of $O$ and its nested objects on the path, in the signature files.

The formal description of the path signature scheme is as follows.

For every class $C$ in a given path of the aggregation hierarchy, there exists a signature file $S$ such that every object $O$ in $C$ has an entry $(\text{sig}, \text{oid\_list})$ in $S$, where $\text{sig}$ is the signature of $O$ and $\text{oid\_list}$ is a list containing the object identifiers of $O$ and its nested objects located on the path.

The signature of an object is created as follows.

1. The signature of a primitive object is generated by hashing the primitive value.
2. The signature of an object not on the path is generated by hashing the object’s OID.
3. The signature of an object on the specified path is generated from superimposing the signatures of all of the object’s attributes.

For a given path in the aggregation hierarchy, files of signatures are created for classes located on the path. As shown in Figure 6, a signature file is associated with
Figure 6: An aggregation hierarchy indexed with path signatures.

Figure 7: Path signature scheme. (a) Format of a signature file; (b) creation of an object signature.
each class on the path Person.Vehicle.Company. Figure 7 shows the signature file for class Person located on the path Person.Vehicle.Company. The signature of Person[i] is the superimposition of the signatures for SSN, Residence, Name, and Owns. The signature of Name is generated by hashing its OID (Lname and Fname are not used). The signature of Owns is generated from all of the primitive values along the path (i.e., Id, Color, Model, Name, and Location).

In the signature file for class Person, the signature of object Person[i] is associated with a list of OIDs, including Person[i] and its nested objects. The purpose of the list is to reduce the expensive traversals between objects. The oid_list includes all nested objects an object directly or indirectly references. Therefore, when a signature matches, we can bypass the intermediate objects and directly retrieve the objects with the specified nested attributes for verification. Actually a tree of object identifiers may be used in the tree signature files. However, the maintenance of a tree of OIDs will be too complex. Since a simple list of OIDs is quite simple to maintain, it is used in the path signature scheme.

3.3 Query Processing and Signature Maintenance

In this section, we describe how two database operations, retrieval and update, are performed with the support of our signature mechanisms. The maintenance of signatures, which is necessary for an update operation, is presented. The signature schemes are general enough to support both queries against a single class and against multiple classes in an inheritance hierarchy. Therefore we don’t explicitly explain the
multiple-class query in our discussion.

3.3.1 Retrieval

Tree Signature

As we mentioned before, there are two approaches to evaluate a query: top-down and bottom-up. The top-down approach is to retrieve all of the objects along the path from the target class to their nested attributes specified in the search condition of the query. Then, the value of the nested attribute is checked to decide if it is a desired object or not. With the signature file, the query is evaluated as follows. A query signature $S_Q$ for the query $Q$ is generated. $S_Q$ is compared with every signature stored in the signature file associated with the target class. If a signature matches with $S_Q$, traverse the path to verify the nested attribute. In the following, the algorithm for top-down retrieval is presented.

**Algorithm** Top-Down Retrieval;

**Given** an object query $Q$,

**retrieve** a set of OIDs specified in the query.

1. Compute the query signature $S_Q$ for the query $Q$.
2. For every entry $(sig_i, oid_i)$ of the signature file associated with the target class, compare $S_Q$ with $sig_i$. If the signatures $sig_i$ and $S_Q$ match, then put $oid_i$ in the returning set $S$.
3. For each object in $S$, traverse the path from the object to the nested attributes specified in $Q$ to eliminate false-drop.
Path Signature

The structure of the path signature file assists the top-down evaluation of the object queries more than the bottom-up evaluation. Therefore, we only present the top-down approach.

Algorithm Top-Down Retrieval;
Given an object query \( Q \), retrieve a set of OIDs specified in the query, in which a nested attribute of class \( C \) is used.

1. Compute the query signature \( S_Q \) for the query \( Q \).

2. For every entry \( (\text{sig}_i, \text{oid} \_\text{list}_i) \) of the signature file associated with the target class, compare \( S_Q \) with \( \text{sig}_i \). If the signatures \( \text{sig}_i \) and \( S_Q \) match, then put the \( \text{oid} \_\text{list}_i \) into a set \( S' \).

3. For each \( \text{oid} \_\text{list}_i \) in \( S' \), retrieve the object of class \( C \) from the list. If its nested attribute satisfies the search condition, put the OID of target object in \( \text{oid} \_\text{list}_i \) into the returning set \( S \).

Since every object on the path is recorded in the \( \text{oid} \_\text{list} \) of the path signature file, there is no traversal along the path for the top-down evaluation. As soon as a matched signature is found, the object which owns the nested attribute may be directly fetched for verification.
3.3.2 Update

Tree Signature

The update operation can be easily done by retrieving the objects for update and then modifying their attributes. However, when we change an attribute of an object, the information in its signature is not accurate anymore. Therefore, we need to update the signature of this object and the signatures of all of its ancestor objects.

To update the signature of a modified object, we have to access all of the directly nested objects of the modified object in order to recompute its signature. If the newly generated signature is the same as the old one, the update process stops. Otherwise, it has to be spread to all of its ancestor objects. Therefore, the two important tasks involving the maintenance of signature files are to compute the signature of an object and to spread the update from the modified object to all of its ancestor objects.

Algorithm compute_signature;

Given the OID of an object O,
compute its signature.

1. Use the OID to retrieve O.

2. Generate the signatures for the primitive attributes.

3. Scan all the signature files of complex attributes to find their signatures.

4. Superimpose the signatures of all of O's attributes.

5. Use OID of O to find the location of its signature in the signature file. Then update the signature.
**Algorithm** spread;

**Given** the OID of an object $O$, which has been updated,

**update** the signatures of $O$ and its ancestor objects.

1. Call `compute_signature` to update the signature of $O$.

2. Stop if the old signature and the new signature are the same.

3. Use the old signature of $O$ to retrieve $O$’s parent objects. (This step will use the retrieval algorithm described in the previous section).

4. Recursively apply spread on all parent objects of $O$.

For this algorithm, if backward references from $O$ to its parent objects are provided, step 3 will be much more efficient.

**Path Signature**

Similar to the tree signature, we have to update the signatures of the modified object and its ancestor objects to maintain the accuracy of the signatures. The maintenance of signatures is divided into two tasks: to compute the signature of an object and to spread the update of the signature from the modified object to its ancestor objects.

**Algorithm** `compute_signature`;

**Given** the OID of an object $O$,

**compute** its signature.

1. Use the OID to retrieve $O$. 
2. If there exists a nested object, \( O' \), for the complex attribute on the path, scan the signature file of \( O' \) to get the signature of \( O' \).

3. Generate the signatures for the remaining attributes by hashing.

4. Superimpose the signature of \( O' \) (if it exists) and the signatures of the other attributes to form the new signature for \( O \).

5. Add \( O \) into the oid_list of \( O' \) and use it as the oid_list for \( O \)'s signature entry.

6. Use OID of \( O \) to find the location of its signature in the signature file. Then update the signature.

Like the tree signature update, the address of the signature may be stored in the object to facilitate direct access of the signature.

The next algorithm is to spread the update of signatures to the classes along the path.

**Algorithm** spread;

**Given** a path of classes, \( C_1, C_2, \ldots, C_n \),

**update** an object \( O \) which belongs to class \( C_i, 1 \leq i \leq n \).

1. Call compute_signature to update the signature of \( O \).

2. If \( i > 1 \), scan through the signature file of class \( C_{i-1} \) to find a set of parent objects which reference \( O \). It can be done by checking if \( O \) is in the oid_list or not.

3. Recursively apply spread on all parent objects of \( O \).
If backward references between objects to their parent objects are provided, the scan of the signature file for parent objects may be avoided.

### 3.4 Cost Models

In this section we formulate a cost model to estimate the storage overhead and performance of the tree signature and path signature schemes. Table 1 lists the parameters and symbols used in the cost model. The unit of size used in the cost model is disk page.

Table 1: Parameters and symbols for cost models of tree and path signature schemes.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>the average number of parents of an object.</td>
</tr>
<tr>
<td>$C$</td>
<td>the average number of classes in the database.</td>
</tr>
<tr>
<td>$E$</td>
<td>the average size of an entry in the signature files.</td>
</tr>
<tr>
<td>$H$</td>
<td>the average height of the aggregation hierarchy for the database.</td>
</tr>
<tr>
<td>$I$</td>
<td>the size of an OID.</td>
</tr>
<tr>
<td>$K$</td>
<td>the average size of a signature file.</td>
</tr>
<tr>
<td>$L$</td>
<td>the average length of the path between target class and a predicate class.</td>
</tr>
<tr>
<td>$N$</td>
<td>the average number of complex attributes in a class.</td>
</tr>
<tr>
<td>$M$</td>
<td>the average number of simple attributes in a class.</td>
</tr>
<tr>
<td>$P$</td>
<td>the average size of an object.</td>
</tr>
<tr>
<td>$Q$</td>
<td>the average number of objects in a class.</td>
</tr>
<tr>
<td>$S$</td>
<td>the size of a signature.</td>
</tr>
<tr>
<td>$R_t$</td>
<td>the average matching rate of a query signature against a tree signature file.</td>
</tr>
<tr>
<td>$R_p$</td>
<td>the average matching rate of a query signature against a path signature file.</td>
</tr>
</tbody>
</table>
3.4.1 Tree Signature

Let $S$ be the size of a signature and $I$ be the size of an OID. The size of an entry in a signature file is:

$$E = S + I.$$  

Thus, the average size of a tree signature file $K$ is:

$$K = E \times Q = (S + I) \times Q$$

where $Q$ is the average number of objects in a class.

**Storage Cost:** The storage overhead for the database is one signature file for each class in the database. Thus, the storage cost is:

$$TSS = K \times C$$

where $C$ is the average number of classes in a database.

**Retrieval Cost:** For a given query, we assume that the length of the path between the target class and the nested attribute is $L$. Then the number of disk page accesses is:

$$TSR = K + R_t \times Q \times L \times P$$

where $R_t$ is the average matching rate of a query signatures against a tree signature file and $P$ is the average size of an object in the database.

**Update Cost:** For an update operation on a particular object, the cost is for the retrieval of the object and the update of the signatures of the modified object and its ancestors. The analysis for recomputation of the signature of an object is as follows.
1. To retrieve the object $O$, $P$ pages of disk access are required.

2. To get the signatures of $O$'s object attributes, $N \cdot K$ disk page accesses are required.

3. To write the signature of $O$ back to its signature file, $K + 1$ page accesses are required, in which $K$ page accesses are for finding the location of $O$'s signature and 1 page access is for writing the updated signature back to the disk.

Therefore, the number of page accesses required to compute a signature is:

$$(N + 1) \cdot K + P + 1.\)$$

The analysis for spreading the update of a signature to its ancestor objects is as follows. The average number of ancestor objects for an object is $A^{H/2}$. To spread the signature recomputation to a parent object of $O$, the old signature of $O$ is used as the key to match its potential parents' signatures. The operation is like a retrieval with a direct attribute in the searching condition. Thus, the cost of disk accesses is:

$$K + R_t \cdot Q \cdot P.$$  

As a result, the total disk accesses for update is:

$$TSU = A^{H/2} \cdot (K + R_t \cdot Q \cdot P + (N + 1) \cdot K + P + 1).$$

### 3.4.2 Path Signature

For a path of $d$ classes $C_1, C_2, ..., C_d$, we estimate the cost for storage, retrieval, and update.
**Storage Cost:** Each object located on the path has a signature entry in one of the signature files. For objects located in $C_i$ where $1 \leq i \leq d$, the size of a signature entry is:

$$S + (d - i + 1) \times I.$$  

There are $Q$ objects in a class. Therefore, the storage overhead is:

$$TSS = \sum_{i=1}^{d} (S + (d - i + 1) \times I) \times Q = d \times S \times Q + Q \times I \times d \times (d + 1)/2.$$  

Therefore, the average size of a path signature file is:

$$K = S \times Q + Q \times I \times (d + 1)/2.$$  

**Retrieval Cost:** For a given query, the number of disk accesses required is:

$$PSR = K + R_p \times Q \times P$$

where $R_p$ is the average matching rate of a query signature against a path signature file.

**Update Cost:** For an update operation, the cost is for the retrieval of the object and the update of the signatures of the modified object and its ancestor. The analysis for recomputation of the signature of an object is as follows.

1. To retrieve the object $O$, $P$ pages of disk access are required.

2. To get the signature of its nested attribute, $K$ page accesses are required.

3. To write the signature of $O$ back to the signature file, $K + 1$ pages are needed, $K$ page accesses for finding the location of $O$'s signature and 1 page access for writing the updated signature back to the signature file.
Therefore, the total number of page accesses required to compute a signature is:

\[ 2 \times K + P + 1. \]

The analysis for spreading the update of a signature to the ancestor objects is as follows. The average number of ancestor objects for an object is:

\[ A^{d/2}. \]

To spread the signature recomputation to the parent objects of \( O \), we check if \( O \) is in the oid list of its potential parents, which requires \( K \) pages of disk accesses. Thus, the total number of disk accesses for update is:

\[ PSU = A^{d/2} \times (3 \times K + P + 1). \]

3.5 Summary

In this chapter, we introduce two signature file methods to support the evaluation of queries on nested objects in OODBs. The filtering capability of our signature schemes will reduce the actual secondary storage accesses and thus improve the performance of object retrieval. Our approaches have a simple file structure and need small storage overheads. The signature schemes not only can support queries with nested attributes, but also can support queries against single or multiple classes in an inheritance hierarchy.

The tree signature scheme generates the signature of an object by hashing all of its direct and nested primitive attributes into a signature. Therefore, the signature is
an abstraction of information directly stored or nested in the object. The path signature scheme generates the signature of an object by hashing all of its direct primitive attributes and nested primitive attributes located on a given path. Consequently, the abstraction of information in the path signature scheme is limited to the primitive attributes along the path. For a given aggregation hierarchy, the information embedded in a path signature is less than that of a tree signature. The cost and complexity of maintenance for the path signature scheme is less than that of the tree signature scheme, because the scope of the index is confined to a path, not the whole aggregation hierarchy. Since there are fewer attributes involved, the path signature scheme is more effective in filtering unqualified objects. On the other hand, the tree signature scheme provides a more general support for queries involving any attribute in the database.
CHAPTER IV

Index Methods for Nested Object Query Processing

4.1 Introduction

The idea behind traditional indexing techniques for nested query processing is to map an attribute value to some ancestor objects which directly or indirectly own the attribute value. The indexing mechanisms implicitly create a direct reverse link from a nested attribute to an ancestor class. As a result, the intermediate objects can be bypassed by scanning indexes. Indexing techniques are effective and will be efficient as long as the overhead they introduce is smaller than the saving gained from avoiding intermediate object traversal. Unfortunately, as pointed out in Section 2.5.1, most indexing techniques require costly storage overhead and expensive index maintenance. Therefore, only some small number of frequently queried target classes and attributes can be chosen to create indexes.

In this chapter, we introduce two new index techniques, called the path dictionary index (PDI) and direct links index (DLI), to support efficient nested query evaluation in object-oriented databases. These two organizations are secondary access structures for object databases. They support associative search on arbitrary number of
attributes with a small amount of storage overhead. The path dictionary index supports efficient evaluation of queries involving any object on a given path of classes. On the other hand, the direct links index supports efficient evaluation of queries involving any object in two arbitrary classes.

We separately describe the concepts and implementations of the path dictionary index and direct links index. Cost models and performance evaluations for these two organizations are also presented.

4.2 Path Dictionary Index

The path dictionary index is a separate access structure for an object database. It consists of two parts: 1) the path dictionary supporting efficient object traversal; and 2) the identity and the attribute indexes supporting associative search. Figure 8 illustrates the overall architecture of the path dictionary index. The identity and attribute indexes are built on top of the path dictionary. Upon the receipt of a query,
the query processor will efficiently evaluate the predicates, if any, using the attribute indexes and then traverse to the target classes using the path dictionary. In other words, the PDI approach reduces the cost of query processing by supporting both associative search and object traversals.

4.2.1 Path Dictionary

![Diagram](image)

Figure 9: (a) A database instance, (b) Path information.

An object-oriented database may be viewed as a space of objects connected with links through complex attributes. Figure 9(a) shows some object instances corresponding to the aggregation hierarchy in Fig. 1. Fig. 9(b) is a *conceptual* path dictionary storing the connections among the objects in the database. Generally speaking, the path dictionary extracts the complex attributes from the database to represent the connections between objects. Since primitive attribute values are not stored in the path dictionary, it is much faster to traverse the nodes in the path dictionary than objects in the database. Therefore, the path dictionary can be used to reduce the
number of accesses to the database, and, in particular, to avoid accessing intermediate objects when traversal from one class to another is performed.

Compared to other approaches which use reverse links from nested attributes to the target objects or store abstract information of (nested) attributes with the objects, the path dictionary prevents unnecessary object accesses by storing the path information among the objects in a separate access structure. The path dictionary provides shortcuts for both forward and backward traversals of the objects on a given path. As a result, it is suitable for general queries, whether they imply top-down or bottom-up evaluation.

When the connections between objects is very complex, the path dictionary can be decomposed into a number of simpler path dictionaries. For instance, a long path may be decomposed into several small path segments for design and efficiency reasons. The configuration issues involved with path and nested indexes were discussed in [14]. In this thesis, we assume that only one path dictionary is built for an aggregation hierarchy.

4.2.2 Attribute Index

While the path dictionary supports fast traversal among objects, it by itself will not help predicate evaluation, which involves finding objects meeting certain conditions specified on their attribute values. To facilitate associative search, the PDI provides attribute indexes which map attribute values to the OIDs in the path dictionary corresponding to the attribute values. As usual, attributes which have high selectivity and are frequently used in queries should be indexed.
Instead of mapping attribute values directly to objects (as in the nested index and path index methods), the attribute indexes map attribute values to path information stored in the path dictionary. The path dictionary serves as a shared structure for object traversal and as a level of indirection from attribute values to the physical objects. The separation of support for traversal and associative search contributes to the low storage overhead and maintenance cost of the PDI approach.

As a result of the separation, as many attributes indexes as necessary can be built on top of the path dictionary without incurring extraordinary growth in storage overhead. The attribute indexes provide general support for various kinds of queries as well as reduce the cost of query evaluation. The more attributes involved in a query, the more options are available for query optimization.

The attribute indexes can be organized as tree-structures, such as B+-trees. However, in order to share the path information with other attribute indexes and to reduce redundant updates on path dictionary, the location of path information in the dictionary should be stored as the leaf nodes of the indexes.

4.2.3 Identity Index

Since OIDs are used to describe the path information among objects, it is often necessary to obtain from the path dictionary path information associated with a given OID. In order to efficiently support this operation, an identity index is provided to map OIDs to the locations in the path dictionary where the OIDs can be found. Since identity search is important for retrieval and update, the identity index significantly
reduces the cost for retrieval and update operations. Similar to the attribute indexes, the identity index is organized as a separate search tree on top of the path dictionary.

4.2.4 Design Considerations

When designing a new access method to support nested query, we considered the following requirements:

- It must be a secondary file organization separately from the databases.
- It must support both forward and backward traversals.
- It must be easily coupled with attribute indexes to support associative search techniques.
- It must support multiple access methods.
- It must support various nested queries efficiently.

The path dictionary index is designed to meet each of the requirements. It is desirable to retain the organization of the original database and keep the index structures transparent to the users, so the access structure is implemented as a secondary file without the need of modifying the database structure at the system level. Also, the access method is general enough to support both forward and backward traversals and a large variety of queries (i.e., the queries we classified in section 1). Since most database queries involve associative search on simple attributes, the method must support both traversal and associative search. The structure is general enough
to support many query evaluation plans for optimization. Finally, the structure is useful for various kinds of queries instead of being useful only for certain kinds of queries. The design of PDI as presented above meets each of these requirements.

4.2.5 s-expression Scheme

In the following, we present an implementation of the path dictionary, the s-expression scheme. The s stands for “subtree”, because each s-expression in the path dictionary stores the path information for a subtree of objects.

The s-expression scheme encodes into an expression all paths terminating at the same object in a leaf class. The s-expression for the path $C_1C_2...C_n$ is defined as follows.

$S_1 = \theta_1$, where $\theta_1$ is the OID of an object in class $C_1$ or null.

$S_i = \theta_i(S_{i-1}[0], S_{i-1}[1]^*)$ 1 < $i$ ≤ $n$, where $\theta_i$ is the OID of an object in class $C_i$ or null, and $S_{i-1}$ is an s-expression for the path $C_1C_2...C_{i-1}$.

$S_i$ is an s-expression of $i$ levels, in which the list associated with $\theta_i$ contains recursively the OIDs of all ancestor objects of $\theta_i$.\footnote{2}$ We call it the ancestor list of $\theta_i$. Except for the objects in $C_1$, every object on the path has an ancestor list, which may be empty.

Note that, by our definition, s-expressions comprise OIDs of objects. However, throughout the thesis, we will use “objects” to refer to the OIDs of objects in the s-expression when it does not cause confusion.

\footnote{1* represents 0 or more $S_{i-1}$.} \footnote{2Although $\theta_i$ denotes the OID of an object, we use it to refer to the object itself, as in this case, when no confusion arises.}
The path dictionary for $C_1 C_2 \ldots C_n$ consists of a sequence of $n$-level s-expressions. The leading object in an s-expression, which does not necessarily belong to $C_n$, is the terminal object of the paths denoted by the s-expression. Thus, the number of s-expressions corresponding to a path equals the number of objects along the path which don't have a nested object on the path. Several s-expressions are shown in Figure 10. They represent the linkage information for the objects on the path Person.Vehicle.Company. In the examples, we use Person[i], Vehicle[i], and Company[i] to refer to the OIDs of the $i$th objects in Person, Vehicle and Company, respectively.

The first s-expression in the figure indicates that there are three paths:

- Person[7].Vehicle[5].Company[1]

all terminating at Company[1], and that Person[3] and Person[7] connect to Company[1] through the common node Vehicle[5]. It is possible that the first $i$ levels of an s-expression are all null, which means the object on level $i+1$ is the terminal object for the subtree represented by the s-expression. For instance, the last s-expression in the figure, ((Person[9])), indicates that Person[9] has no car and therefore no manufacturer for the car. On the other hand, an s-expression may contain null ancestor lists.
indicating that the object is not referenced by any other object. For instance, in
the third s-expression in Fig. 10, the ancestor list for Vehicle[3] is empty, meaning
that the vehicle doesn't have an owner. An advantage of the s-expression scheme is
that every object on the path appears only once in the path dictionary, thus avoiding
redundant partial path information introduced in other schemes.

Due to the inherent tree structure, the s-expression scheme supports naturally 1:1
and 1:N relationships. N:M relationship can be easily supported by extending the s-
expression. The analysis in this thesis, however, is based on 1:1 and 1:N relationships,
because these are most common in database applications. Furthermore, since the goal
of the analysis is to compare the performance of PDI against the nested and path
index methods, given the decoupling of traversal and associative search support the
performance advantage of PDI for N:M relationships is even more pronounced than
that of 1:1 and 1:N relationships.

4.3 Implementation of the s-expression Scheme

Figure 11(a) illustrates the data structure of an s-expression for \(C_1C_2...C_n\). \(SP_i\) in
the header represents the starting position for locating objects of class \(C_i\) in a s-
expression. It points to the first occurrence of \(\theta_i\) in the s-expression. Following the
\(SP_i\) fields is a series of \((OID, Offset)\) pairs. At the end of the s-expression is a special
end-of-s-expression (EOS) symbol. The data structure mimics the nesting structure
in the s-expression. The OIDs in the data structure are in the same order as the
OIDs in the s-expression. The offset associated with \(\theta_i, 2 \leq i \leq n\), points to the
next occurrence of $\theta_i$ in the $s$-expression. The OIDs for class $C_1$ don’t have offset fields, since all $\theta_1$’s referencing the same $\theta_2$ are stored consecutively right after $\theta_2$; for the same reason, $SP_1$ is not needed either, since $\theta_1$’s can be located by tracing $\theta_2$’s. Using the $SP_i$ and offset values, we can easily trace the nested relationship among objects in an $s$-expression. For example, to obtain the ancestor list associated with $\theta_i$, we simply collect the OIDs stored after $\theta_i$ until we reach the OID pointed to by $\theta_i$’s offset. An $s$-expression and its representation are shown in Fig. 11(b) and (c), respectively.

An advantage of this representation is that it allows fast retrieval of OIDs in the same class. To retrieve all OIDs for class $C_i$, we start with $SP_i$, which will lead us to the first $\theta_i$ in the $s$-expression. Following the associated offset value we can reach the next $\theta_i$, and so on. Thus, we can quickly scan through all OIDs in a class, skipping
the OIDs of irrelevant classes. Notice that the offset associated with $\theta_n$ is pointing to $\theta_n$ in the next s-expression, because there is at most one OID of class $C_n$ in an s-expression.

S-expressions are stored sequentially on disk pages. In order to reduce the number of page accesses, an s-expression is not allowed to cross page boundaries unless the size of the s-expression is greater than the page size. If an s-expression is too long to fit into the space left in a page, a new page is allocated. Consequently, free space may be left in a page. Updates and insertions may cause a page to overflow, which requires a new page to be allocated and some of the s-expressions in the overflown page to be moved to the new page. In order to effectively keep track of the free space available in the pages, a free space directory (FSD), which records the pages with free space above a certain threshold, is maintained at the beginning of the path dictionary.

Figure 12(a) illustrates the physical structure of the path dictionary index, which consists of the free space directory, the s-expression pages, and attribute and identity indexes. Fig. 12(b) and (c) show the structures of a leaf node record and a non-leaf node for B+-tree implementation of the identity index, respectively. In the identity index, the OIDs are used as the key value. The s-address in the leaf node is the address of the s-expression corresponding to the OID in the same leaf node. The page pointers in a nonleaf nodes are pointing to the next level of nonleaf nodes or to the leaf nodes.

Figures 12(d) and (e) show the structures of an attribute index's leaf node record and nonleaf node page. The OIDs and s-expression addresses (denoted as s-addr)
Figure 12: Path dictionary index. (a) Structure of the path dictionary index; (b) leaf node record of the identity index; (c) nonleaf node of the identity index; (d) leaf node record of an attribute index; (e) nonleaf node of an attribute index.
are used to access the s-expressions of the corresponding OIDs. Attribute indexes improve the path dictionary’s performance in predicate evaluation and range query processing, because single-value predicates and range predicates can be performed by efficient index scanning rather than accessing all of the objects in the predicate classes.

4.4 Retrieval and Update with Path Dictionary Index

In the following, we discuss the strategies used to process nested queries and updates with the path dictionary index.

4.4.1 Retrieval Operations

In order to simplify our discussion, we assume that the query has only one predicate attribute, which is indexed by an attribute index, $Index_p$. We specify a nested query $Q$ as having $C_t$ as the target class and $C_p$ as the predicate class, where $1 \leq t, p \leq n$. We use $\theta_t$ and $\theta_p$ to denote OIDs of objects in class $C_t$ and $C_p$.

We assume that a path dictionary index for the path $C_1C_2...C_n$ has been created. $Index_p$ is the attribute index based on an attribute of $C_p$. The path dictionary supports all classes of the nested queries. The following strategies are applicable to both TP and PT queries.

**Simple Predicates:** Attribute indexes are advantageous for processing simple predicates with equality and range operations. For inequality operation, the path dictionary is still better than the conventional traversal approaches.
Equality — We use the attribute value specified in the predicate to search attribute index $Index_p$ for the corresponding addresses of the s-expressions. Through the addresses, we can obtain the s-expressions and derive from the s-expressions the OIDs for $C_t$. PDI allows us to avoid accessing any objects from the database.

For example, assuming that the attribute Name of Company is indexed by $Index_{name}$, we can answer the query Q1: “retrieve persons who own cars made by GM” by first searching $Index_{name}$ using “GM” as the search key to obtain the addresses of the s-expressions corresponding to “GM”. After the s-expressions are accessed through the addresses, the OIDs of the Person objects in the s-expressions are returned.

Range — For the range query, we use the lowest key value in the range to search the attribute index for the leaf node record containing the lowest key value. Then we sequentially search the leaf node records until the record containing the highest key value in the range is reached. From those leaf node records, we obtain the addresses of the s-expressions corresponding to the predicate objects with an attribute value in the specified range. As before, we obtain the s-expressions and return the OIDs for $C_t$ from the s-expressions. This strategy prevents repeated scanning on the attribute index.

For example, assuming that the attribute Age of Person is indexed by $Index_{age}$, we can answer the query Q5: “retrieve manufacturers of the
cars owned by persons older than 50” by using “50” as the search key on Index$_{age}$ to arrive at the leaf node record corresponding to 50. Starting from the next leaf node record, which is corresponding to the next age greater than 50, we sequentially scan the leaf nodes and use the addresses in the records to access the corresponding s-expressions. From the s-expressions, the OIDs of the Company objects are returned. In this example, the scanning of the leaf nodes continues until there are no more leaf node records left.

**Inequality** — The attribute indexes cannot be used for predicates with inequality operation. However, the path dictionary can still improve the processing of this class of queries. The objects in $C_p$ are retrieved from the database for predicate evaluation. The OIDs of the qualified objects, $\theta_p$’s, are collected in a set $P$. The OIDs in $P$ are used as keys to search the identity index for the s-expression addresses. Then the s-expressions in the path dictionary are accessed to return the OIDs of the qualified objects in the target class (i.e., $\theta_t$’s), with which the target objects can be retrieved from the database. Using the path dictionary, we avoid accessing from the database any objects between $C_t$ and $C_p$.

For example, to answer the query Q3: “retrieve persons who own cars not made by GM”, all of the objects in class Company are accessed to collect OIDs of GM company objects into a set $P$. The OIDs in $P$ are then used as keys to search the identity index for s-expressions. Finally the OIDs
of objects in the target class are derived from the found s-expressions and returned.

**Complex predicates:** Attribute indexes have great advantages on predicate evaluation. Unfortunately, they don’t benefit queries with complex predicates, which require scanning the identity index or sequentially searching the path dictionary. The strategies for answering this class of nested queries are different depending on the existence of $\theta_p$ in the predicate.

**Exist** — If $\theta_p$ is specified in the predicate, we can use the identity index to locate the s-expressions containing $\theta_p$ from the path dictionary and derive $\theta_t$ from the s-expression for predicate evaluation. If the relationship between $\theta_p$ and $\theta_t$ satisfies the predicate, $\theta_t$ is returned.

For example, to answer the query Q7: “retrieve persons who have a car made by Company[1]”, we search the identity index using Company[1] as the key, obtain from the path dictionary the s-expressions corresponding to Company[1], and derive from the s-expressions the OIDs of Person objects.

**Nonexist** — If no $\theta_p$ is specified in the predicate, we will scan the path dictionary for the s-expressions in which the predicate on $C_t$ and $C_p$ is satisfied, and return $\theta_t$.

For example, to answer the query Q8: “retrieve persons who don’t have a car”, the s-expressions in the path dictionary are sequentially searched for the pattern “((Person?))” and the matching Person objects are returned.
Without the path dictionary, we will have to examine every `Person` object in the database and check if the `Owns` attribute is null or not.

On the other hand, to evaluate Q10: "retrieve vehicles which are not owned by any person" (the PT case), we sequentially scan the path dictionary and simply return all of the vehicle objects with an empty ancestor list (i.e., vehicle objects matching the pattern "vehicle()"). Without the path dictionary, the query would be very expensive since it requires a scan through the `Vehicle` class to collect all OIDs in it, another scan through the `Person` class to collect all OIDs under the `Owns` attribute (i.e., all vehicles with owners), and a set difference between the two result sets.

For queries with predicates on more than one indexed attribute, the evaluation is accomplished by first separately scanning the attribute indexes, with the results unioned or intersected according to the Boolean condition in the query. For each index, addresses of the s-expressions, corresponding to objects which passed the predicates, are collected. Next, these sets of s-expression addresses are unioned or intersected to generate a set $S$ in accordance with the Boolean combination of the search conditions. Use addresses in $S$ to access the s-expressions. The OIDs of objects corresponding to unindexed predicate classes are derived from the s-expressions and used to access the objects in the database. After the evaluation of the search condition is completed, the OIDs of the qualified objects in target class are returned from the s-expression.
Compared to the nested index, the path dictionary index approach needs to derive the target objects from the \textit{s}-expressions in the path dictionary, while the nested index will directly return the qualified target objects through index scan. However, with queries involving unindexed attributes, the nested index needs to traverse the objects in the database in order to evaluate the predicate, while the path dictionary index can directly access the objects in the target class. Besides, with queries involving more than one indexed attribute, the cost of index scans is about the same for both methods. Although the path dictionary index may cost more when the number of \textit{s}-expressions accessed is large, we expect the path dictionary index to have about the same retrieval performance as the nested index for queries with typical selectivity and reasonably restricted conjunctive conditions.

4.4.2 Update Operations

When changes are made to the database, the path information in the dictionary must be updated. Operations such as update, insert, delete, create, and destroy will require updates to the path dictionary. In the thesis, we only describe the update operation.

When changes are made to the database, the path information in the dictionary must be updated. Since updates to simple attributes won't change the links among objects, they have no effect on the path dictionary. When complex attributes are modified, however, the path dictionary must be updated. However, owing to the attribute indexes, updates on the simple attributes of the objects located along the indexed path induces updates on the PDI. The PDI has to be updated in the following situations:
1. When an indexed simple attribute is modified: the corresponding attribute index has to be updated, while the path dictionary and the identity index need not be changed. Suppose one of the indexed attributes of an object, identified by \( \theta \), is modified. Let \( A_{\theta} \) be the address of the s-expression containing \( \theta \). The update of the attribute indexes is accomplished by two index scans: one to delete \( A_{\theta} \) from the leaf node corresponding to the old attribute value, and the other to insert \( A_{\theta} \) in the leaf node corresponding to the new attribute value.

2. When one of the complex attributes connecting the path is modified: Suppose object \( O_i \) changes its complex attribute from \( O_{i+1} \) to \( O'_{i+1} \) (\( O_i, O_{i+1} \) and \( O'_{i+1} \) are identified by \( \theta_i, \theta_{i+1} \) and \( \theta'_{i+1} \).) If none of the direct attributes of class \( C_i \) and none of the direct attributes of \( C_i \)'s ancestor classes are indexed, we have to search the path dictionary through the identity index to find the s-expressions containing \( \theta_i \) and \( \theta'_{i+1} \). Then, \( \theta_i \) and its ancestor list are moved from the ancestor list of \( \theta_{i+1} \) to the ancestor list of \( \theta'_{i+1} \). Meanwhile, the identity index has to be updated by changing the old s-expression address in \( \theta_i \)'s leaf node to the new address. However, if some direct attributes of class \( C_i \) or \( C_i \)'s ancestor classes are indexed, we also have to update those attribute indexes, which is the same as described above.

Note that an alternative approach is to traverse from \( O_{i+1} \) and \( O'_{i+1} \) to their nested attributes, then use the attribute values to scan through the attribute index to locate the s-expressions and update the path dictionary. The update of the attribute index may be done while locating the s-expression addresses.
Whether this method is better than the previous one depends on whether or not traversing through the nested attributes is more expensive than going through the identity index.

Let’s consider the following update examples. Assume that the attribute \textit{Age} of class \textit{Person} is indexed by \textit{Index\textsubscript{age}}. The update “change \textit{Person}[1]’s age from 50 to 51” will not change the path dictionary and the identity index, but \textit{Index\textsubscript{age}} has to be searched twice to move \textit{Person}[1]’s s-expression address from the leaf node corresponding to 50 to the leaf node corresponding to 51. Next, for the update “change \textit{Person}[1]’s car from \textit{Vehicle}[7] to \textit{Vehicle}[10]”, we first use the identity index to locate the s-expressions corresponding to \textit{Vehicle}[7] and \textit{Vehicle}[10]. The path dictionary is updated by moving \textit{person}[1] from the s-expression corresponding to \textit{Vehicle}[7] to the s-expression corresponding to \textit{Vehicle}[10]. The identity index is then updated by changing the leaf node of \textit{Person}[1] from pointing to the s-expression corresponding to \textit{Vehicle}[7] to that corresponding to \textit{Vehicle}[10]. Finally, the attribute index \textit{Index\textsubscript{age}} has to be updated by removing the s-expression address corresponding to \textit{Vehicle}[7] and inserting the s-expression address corresponding to \textit{Vehicle}[10] into the leaf node of \textit{Index\textsubscript{age}} corresponding to the age of \textit{Person}[1].

4.5 Storage Cost and Performance Evaluation

In this section, we formulate the cost models for the path index, path dictionary, and path dictionary index methods to analyze their storage overhead and query processing performance. Then, we compare their performance in terms of their storage, retrieval,
and update costs. We select the path index [11] as a reference point in the comparison, because it can be generally applied to queries with different target classes as long as the classes are on the indexed path (i.e., TP queries). However, the path index can't be used for PT queries, because its structure implies a bottom-up evaluation. The path dictionary and path dictionary index are general enough to provide significant support for both TP and PT queries. The path dictionary method can serve as the baseline performance for the path dictionary index method, where no attribute index is used.

Table 2 lists the parameters and symbols we used in development of cost models for path dictionary and path dictionary index. In order to facilitate our comparison with path index, we adopt some common parameters from [11]. The parameters are used to describe the characteristics of the classes and their attributes on the path, $C_1C_2...C_n$, and the structures of the three organizations.

Performance is measured by the number of I/O accesses. Since a page is the basic unit for data transfer between main memory and external storage, we use it to estimate the storage overhead and the performance cost. All lengths and sizes used above are in bytes.

In the cost models and our comparisons, the ratios of shared references and shared attribute values play an important role. The ratio of shared references for a class $C$ and its domain class $D$ refers to the average number of different objects in $C$, which have the same object in $D$ as their complex attribute value. Likewise, the ratio of shared reference values for a class $C$ and a simple attribute $A$ refers to the average
Table 2: Parameters and symbols for cost models of path index and path dictionary index.

- \( N_i \): the number of objects in class \( C_i \), \( 1 \leq i \leq n \).
- \( S_i \): the average size of an object in class \( C_i \).
- \( A_i \): the complex attribute of \( C_i \) used on the path, \( 1 \leq i \leq n \).
- \( D_i \): the number of distinct values for complex attribute \( A_i \).
- \( k_i \): the ratio of shared reference between objects in class \( C_i \) and values for \( A_i \). \( (k_i = N_i/D_i) \)
- \( A_{ij} \): the \( j \)th simple attribute of \( C_i \), \( 1 \leq i \leq n \).
- \( U_{ij} \): the number of distinct values for simple attribute \( A_{ij} \) of class \( C_i \).
- \( q_{ij} \): the ratio of shared attribute value between objects in class \( C_i \) and values for attribute \( A_{ij} \). \( (q_{ij} = N_i/U_{ij}) \)
- \( K \): the average ratio of shared references, i.e., \( k \)'s, and shared attribute values, i.e., \( q \)'s.
- \( UIDL \): the length of an object identifier.
- \( P \): the page size.
- \( pp \): the length of a page pointer.
- \( f \): average fanout from a nonleaf node in the path index, identity index, and attribute indexes.
- \( kl \): average length of a key value in path index and attribute indexes.
- \( ol \): the sum of the key length, record length, and number of path fields in the path index.
- \( OFFL \): the length of an offset field in the path dictionary.
- \( SL \): the length of the start field in the path dictionary.
- \( FSL \): the length of the free space field in the free space directory.
- \( EL \): the length of EOS.

number of different objects in \( C \) which have the same attribute value for \( A \). Also, the fanout represents the number of pointers for the next level of nonleaf nodes in a tree-structured index.

To directly adopt the formulae developed in [11], we follow their assumptions:

1. There are no partial instantiations, which implies that \( D_i = N_{i+1} \).
Table 3: Parameter values for the cost models of path index and path dictionary index.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>4096</td>
</tr>
<tr>
<td>$UIDL$</td>
<td>8</td>
</tr>
<tr>
<td>$pp$</td>
<td>4</td>
</tr>
<tr>
<td>$OFFL$</td>
<td>2</td>
</tr>
<tr>
<td>$SL$</td>
<td>2</td>
</tr>
<tr>
<td>$FSL$</td>
<td>2</td>
</tr>
<tr>
<td>$EL$</td>
<td>4</td>
</tr>
<tr>
<td>$kl$</td>
<td>8</td>
</tr>
<tr>
<td>$ol$</td>
<td>6</td>
</tr>
<tr>
<td>$f$</td>
<td>218</td>
</tr>
</tbody>
</table>

2. All key values have the same length.

3. Attribute values are uniformly distributed among the objects of the class defining the attribute.

4. All attributes are single-valued.

Further, we adopt the parameter values in [11]. Table 3 lists the values chosen for the path index and the path dictionary index.

### 4.5.1 Storage Overhead

In this section, we present the cost models for storage overhead of the path index, path dictionary and path dictionary index. The formulae for path index is adopted from [11]. Then, we develop formulae for the path dictionary. Since the path dictionary is also a part of the path dictionary index, we further derive the formulae for storage overhead of the attribute indexes. The overall storage overhead for path dictionary index is the sum of path dictionary and attribute indexes. Finally, by using the cost models developed, we compare the storage overhead of the path index, path dictionary and path dictionary index.
Path Index

To create a path index for a primitive attribute, $A_{n,j}$, of the class $C_n$, which maps the key values of $A_{n,j}$ to every class on the path $C_1C_2...C_n$, the number of leaf node pages needed is as follows[11]. Let $PN$ be the ratio of shared references for the class $C_i$ and the attribute $A_{n,j}$. Thus, $PN = k_1k_2...k_{n-1}q_{n,j}$. Let $XP$ be the average size of leaf nodes for the path index.

If $XP \leq P$:

$$LP = \lceil U_{n,j}/[P/XP] \rceil,$$

where $XP = PN \cdot UIDL \cdot n + kl + ol$.

If $XP > P$:

$$LP = U_{n,j}[XP/P],$$

where $XP = PN \cdot UIDL \cdot n + kl + ol + [(PN \cdot UIDL \cdot n + kl + ol)/P] \cdot (UIDL \cdot n + pp)$.

The number of nonleaf pages is:

$$NLP = \lceil LO/f \rceil + \lceil [LO/f]/f \rceil + ... + X,$$

where $LO = \min(U_{n,j}, LP)$ and $X < f$. If $X \neq 1$, $NLP$ is increased by one to account for the root node. The total number of pages needed for the path index is:

$$PIS = LP + NLP.$$ (4.1)
Path Dictionary

The path dictionary (PDI) consists of three parts: s-expressions, free space directory and identity index. Therefore, we derive formulae for these three parts as follows.

Each object in the path dictionary, except for those in the root class of the path, is associated with an offset. Therefore, an object will take at most \((UIDL + OFFL)\) bytes in an s-expression. The average number of objects in an s-expression is:

\[
NOBJ = 1 + K_{n-1} + K_{n-1}K_{n-2} + ... + K_{n-1}K_{n-2}...K_1.
\]

Thus, the average size of an s-expression is:

\[
SS = SL \cdot (n - 1) + (UIDL + OFFL)NOBJ + EL.
\]

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

\[
SSP = \begin{cases} 
\left\lfloor \frac{N_n}{\lfloor P/SS \rfloor} \right\rfloor & \text{if } SS \leq P \\
N_n \left\lfloor SS/P \right\rfloor & \text{if } SS > P.
\end{cases}
\]

In order to efficiently manage the free spaces in the path dictionary, a free space directory is maintained in the top of the path dictionary (see Figure 12). The number of pages needed for the free space directory is:

\[
FSD = \left\lfloor SS(PP + FSL)/P \right\rfloor.
\]

Next, we derive the storage overhead for the identity index. The total number of objects in the database is:

\[
TOBJ = N_1 + N_2 + ... + N_n = NOBJ \cdot N_n.
\]
The number of leaf pages needed for the identity index of the path dictionary is:

\[ LP_{\text{identity}} = \lceil \text{TOBJ}/(P/(UIDL+pp)) \rceil. \]

The number of nonleaf pages is:

\[ NLP_{\text{identity}} = \lceil LP_{\text{identity}}/f \rceil + \lceil LP_{\text{identity}}/f \rceil + \ldots + \lfloor X \rfloor, \]

where \( X < f \). If \( X \neq 1 \), \( NLP_{\text{identity}} \) is increased by 1 to account for the root node.

The total number of pages needed for the identity index is the sum of nonleaf pages and leaf node pages:

\[ IIP = LP_{\text{identity}} + NLP_{\text{identity}}. \]

Therefore, the number of pages needed for the path dictionary is:

\[ PDS = FSD + SSP + IIP. \] (4.2)

**Path Dictionary Index**

The path dictionary index consists of a path dictionary and attribute indexes. Since the dictionary part of the path dictionary index is exactly the same as the path dictionary, the number of pages for the dictionary part of PDI is the same as that of the path dictionary we derived above. In the following, we further analyse the attribute index part of PDI.

For an attribute index based on the \( j \)th primitive attribute, \( A_{i,j} \), of the class \( C_i \), the average number of pages needed for a leaf node record is:

\[ XP_{A_{i,j}} = kl + ol + q_{i,j}(UIDL+pp). \]
The number of leaf node pages needed is:

\[ LP_{Ai,j} = \begin{cases} 
\frac{U_{i,j}}{P/XP_{Ai,j}}, & \text{if } XP_{Ai,j} \leq P \\
U_{i,j}\left[\frac{XP_{Ai,j}}{P}\right], & \text{if } XP_{Ai,j} > P.
\end{cases} \]

The number of nonleaf pages is:

\[ NLP_{Ai,j} = \left\lfloor \frac{LO_{Ai,j}}{f} \right\rfloor + \left\lfloor \frac{\left\lfloor \frac{LO_{Ai,j}}{f} \right\rfloor}{f} \right\rfloor + ... + X, \]

where \( LO_{Ai,j} = \min(U_{i,j}, LP_{Ai,j}) \) and \( X < f \). If \( X \neq 1 \), \( NLP_{Ai,j} \) is increased by one to account for the root node. Thus, the total number of pages needed for indexing \( A_{i,j} \) is:

\[ AIP_{Ai,j} = LP_{Ai,j} + NLP_{Ai,j}. \]

Assuming that \( m \) attribute indexes, \( index_1, index_2, ..., index_m \), are created for the path dictionary index, the number of pages needed for the path dictionary index is:

\[ PDIS = PDS + AIP_{index_1} + AIP_{index_2} + ... + AIP_{index_m}, \]  

(4.3)

where \( index_1, index_2, ..., index_m \) are the attribute indexes created.

**Comparison**

Using the formulae developed above, we compare the storage overhead of the path index, path dictionary, and path dictionary index. We choose a path of 4 classes in the comparison. For the path index and path dictionary index, a primitive attribute in the bottom class of the path, \( A_{4,1} \), is chosen for indexing. Also, we fix the cardinality of \( N_1 \) to 200000 and the average size of an object to 80 bytes. In the following, we use \( PIS, PDS \) and \( PDIS \) to represent the storage overhead of the path index, path dictionary and path dictionary index, respectively.
To observe the impact of the ratios of shared references and shared key values on the storage overhead, we vary the average ratio, $K$, from 1 to 25. Figure 13 shows that $PIS < PDS < PDIS$ when $K = 1$ and 2, and that $PDS < PDIS < PIS$ when $3 \leq K \leq 25$. The zigzag lines in the figure are caused by the fragmenting of disk pages. The explanation for the case of $K = 1$ is that when there are no shared references and key values in the database, the structure of an $s$-expression in the path dictionary methods is similar to that of a leaf node record in the path index, except that the path dictionary methods have additional storage overhead for the offset fields in the $s$-expressions, the identity index, and the leaf node records in the attribute indexes. However, the amount of redundant path information in the path index increases when the ratios of shared references and shared key values
increase. Therefore, we can conclude that, in general, the path dictionary and the path dictionary index have better storage overhead than the path index.

In practice, we usually have more than one attribute in the path to be indexed. In order to compare the overall storage overhead for the indexes created, we calculate the total cost of creating $n$ indexes on attributes of class $C_4$ for the path index and the path dictionary index. We vary $n$ from 1 to 10 to observe the change of storage overhead for the three methods. In this comparison, we fix the ratios of the shared references among the classes along the path and the ratios of the shared key values for each attribute indexes (i.e., $k$’s and $q$’s are set to 3).

Figure 14: Storage overhead for multiple attribute indexes ($K = 3$).
Figure 14 shows that \( PIS \) increases dramatically, because the path index creates a separate index for each attribute indexed. \( PDS \), shown here as a reference, is constant since it doesn't create any index. On the other hand, the increase on \( PDIS \) is due only to the storage for building the attribute indexes. Thus, we can see that \( PDIS \) increases as the number of attribute indexes increase, but at a much lower rate than \( PIS \).

### 4.5.2 Retrieval Cost

In the following, we present the cost models for retrieval cost of the path index, path dictionary and path dictionary index. Like the previous section, we adopt the formulae for path index from [11] and develop formulae for the path dictionary and path dictionary index.

Using the classification introduced in Section 2.3 queries on nested objects can be classified as TP, PT and MX. To simplify our analysis, we assume that there is only one predicate attribute in the queries. Therefore, we will only consider TP and PT queries in our discussion.

**Path Index**

Since the structure of the path index implies a bottom-up evaluation, it can't be applied to PT queries. Therefore, the traditional forward traversal approach is used. The cost model for evaluating TP queries with the path index is given in [11]. The number of pages accessed for retrieval is:

\[
PIR = \begin{cases} 
  h + 1 & \text{if } XP \leq P \\ 
  h + \lfloor XP/P \rfloor & \text{if } XP > P,
\end{cases}
\]  

(4.4)
where \( h \) = height of the path index – 1, and \( XP \) is the size of a leaf node record in the path index.

**Path Dictionary**

The path dictionary may be applied to \( TP \) and \( PT \) queries. To answer a query \( Q \) which has \( C_t \) as the target class and \( C_p \) as the predicate class, where \( 1 < t, p < n \), the path dictionary approach will have to retrieve all of the objects in class \( C_p \) for predicate evaluation, search the identity index to locate the addresses of the s-expressions, then access the s-expressions in the path dictionary to return the objects in \( C_t \). Therefore, the number of pages accessed is:

\[
PDR = \left\lfloor \frac{N_p S_p}{P} \right\rfloor + N_{p|Q}(h_{identity} + 1 + \left\lfloor SS/P \right\rfloor),
\]

where \( N_{p|Q} \) is the number of objects in class \( C_p \), which satisfy the predicates in \( Q \), and \( h_{identity} \) = height of the identity index – 1.

**Path Dictionary Index**

Likewise, the path dictionary index supports both \( TP \) and \( PT \) queries. To answer \( Q \) using the path dictionary index, we need to traverse a number of nonleaf nodes and one leaf node record in the attribute index, \( Index_{A_i,j} \), and then access the path dictionary. Therefore, the number of pages accessed is:

\[
PDIR = h_{attr.} + \left\lfloor \frac{XP_{attr.}}{P} \right\rfloor + N_{p|Q} \cdot \left\lfloor SS/P \right\rfloor
\]

where \( h_{attr.} \) = height of the attribute index – 1.
Comparison

In the following, we conduct two comparisons among the path index, path dictionary and path dictionary index. We first evaluate the retrieval cost for a single TP query which is specially suitable for path index (since the path index can only support the TP queries). Then we evaluate the average retrieval cost of a variety of queries.

A single TP query: We use the same parameters and assumptions as we used in evaluating the storage cost. We use $PIR, PDR$ and $PDIR$ to represent the retrieval costs of the path index, the path dictionary and the path dictionary index, respectively.

First, we assume that the query has $C_1$ as the target class, $C_4$ as the predicate class, and $A_{4,1}$ as the indexed predicate attribute. We assume that all $k$ and $q$ values equal to an average ratio, $K$. As before, we increase $K$ from 1 to 25 to observe the effect on retrieval cost.

Figure 15 indicates that the path index has the lowest retrieval cost initially, while the path dictionary index is a close second. However, as $K$ increases, the effect of redundant path information in the path index becomes dominant, costing more page accesses. After $k = 5$, the path dictionary index has a lower retrieval cost than the path index; after $k = 10$, the path dictionary also has a lower retrieval cost than the path index. Therefore, we may conclude that for a query with an indexed attribute in the predicate:

- $PIR < PDIR < PDR$ when $K < 5$
- $PDIR < PIR < PDR$ when $5 \leq K < 10$
- $PDIR < PDR < PIR$ when $10 \leq K$
Mix of TP queries: The path dictionary and the path dictionary index are more general mechanisms than the path index in terms of improving the overall performance for different kinds of queries. The path dictionary mechanisms may be used to process any kind of queries which have predicate attributes in the classes located along the path. The path index, however, can only be used to process queries with predicates on the indexed attributes.

To compare the overall retrieval performance of the three methods, we select the following mix of queries for evaluation:

1. Three queries in which the indexed attribute, \( A_{4,1} \), of \( C_4 \) is the only predicate attribute and \( C_1, C_2 \) and \( C_3 \) are the target classes, correspondingly.
2. Three queries in which a non-indexed attribute, \( A_{4,2} \), of \( C_4 \) is the predicate attribute and \( C_1, C_2 \) and \( C_3 \) are the target classes, correspondingly.

3. Two queries in which \( C_2 \) and \( C_3 \) are the predicate classes, correspondingly, and \( C_1 \) is the target class.

Note that we only include the TP class of the queries in the list. Since the queries in 2 and 3 are not supported by the path index, we have to use the traditional forward traversal or backward traversal approaches to evaluate these queries. When the queries are not supported by the path index, we use the cost models for retrieval without path index/path dictionary developed in [35] to compute the retrieval cost.

![Figure 16: General retrieval cost for mixed TP queries.](image)
As before, we vary the average ratio of shared reference and shared key values, $K$, from 1 to 25 to observe the retrieval performance of the methods with respect to $K$. Figure 16 shows that $PDR$ and $PDIR$ have a much better overall retrieval performance than $PIR$. The overall performance of $PDIR$ will be better if we index more attributes on the path. Likewise, the overall retrieval performance of the database will improve if we create more path indexes on different attributes. However, some queries, such as PT queries, won't be supported at all by the path index method. Also, the cost of building more path indexes is very expensive as shown previously.

4.5.3 Range Query Cost

Range query is one of the important operations for database query. In this section, we develop cost models for the path dictionary and the path dictionary index. Using the formulae, we compare the cost of a range query with respect to the percentage of objects satisfying the query predicate.

Path Index

Based on [11], the number of page accesses is:

$$PIRQ = \begin{cases} h + \left\lceil \frac{NRQ}{NREC} \right\rceil & \text{if } XP \leq P \\ h + \left\lceil \frac{NRQ}{[XP/P]} \right\rceil & \text{if } XP > P, \end{cases}$$

(4.7)

where $h = \text{height of the path index} - 1$, $XP$ is the size of a leaf node record for the path index, $NRQ$ is the number of key values specified in the query, and $NREC$ is the number of records in a leaf node page.
Path Dictionary

For range queries on the path dictionary, the formula is the same as the formula for single value retrieval:

\[ PDRQ = \lceil N_p S_p / P \rceil + N_p|Q(h + 1 + \lceil SS/P \rceil), \]

(4.8)

where \( N_p|Q \) is the number of objects in class \( C_p \) satisfying the predicates in \( Q \) and \( h = \text{height of the identity index} - 1 \).

Path Dictionary Index

To answer a range query with the path dictionary index, an attribute index scan is performed to find the leaf node record corresponding to the lower bound value of the range. Then, sequential access to the leaf node records satisfying the query range is made. Finally, all of the s-expressions specified in these records are retrieved in order to return the target objects. Thus, the number of pages accessed is:

\[ PDIRQ = \begin{cases} h + \lceil N_RQ / NREC \rceil + N_p|Q \lceil SS/P \rceil & \text{if } XP \leq P \\ h + \lceil N_RQ / [XP/P] \rceil + N_p|Q \lceil SS/P \rceil & \text{if } XP > P \end{cases}, \]

(4.9)

where \( h = \text{height of the attribute index} - 1 \), \( XP \) is the size of a leaf node record for the attribute index, \( N_RQ \) is the number of key values specified in the query, and \( NREC \) is the number of records in a leaf node page.

Comparison

Instead of varying the ratio of references between classes, we change the selectivity of the predicate specified in the query, \( RANGE \), from 1% to 100%. In the comparison, we fix \( k_1, k_2, k_3 \) and \( q_{4,1} \) to 3. The number of key values satisfying the predicate, \( N_RQ \),
is $U_{4,1}\cdot RANGE$. Therefore, the number of objects in $C_4$ satisfying the query predicate, $N_{pl/Q}$, is $NR_Q\cdot q_{4,1}$. $N_{pl/Q}$ is used in computing the cost of range queries for the path dictionary and the path dictionary index.

Figure 17 shows that the path index has a much better performance on answering range queries than the other two approaches. This is because the leaf node records of the path index are sorted based on the key values of the indexed attribute. After accessing the record corresponding to the lowest key value, the OIDs of the target objects may be returned by sequentially scanning the leaf nodes until the highest range value is reached. On the other hand, the path dictionary index has to access s-expressions in the path dictionary to return the OIDs of the target objects after
they obtain the addresses of s-expressions. The path dictionary approach has the worst performance, because the evaluation of predicates is based on accessing the objects in the predicate class.

### 4.5.4 Update Cost

In the following, we present the cost model for an update operation. Other update operations, such as insertion and deletion, may be derived in a similar way. To simplify the analysis, we do not include the cost due to page overflow caused by insertion or update operations. In the following, we assume that the complex attribute $A_{i+1}$ of $O_i$ is changed from $\theta_{i+1}$ to $\theta'_{i+1}$ ($\theta_{i+1}$ and $\theta'_{i+1}$ are OIDs of $O_{i+1}$ and $O'_{i+1}$).

**Path Index**

Suppose that the path index is based on the simple attribute, $A_{n,1}$ of class $C_n$. To determine the nested attribute values in $A_{n,1}$ for $O_{i+1}$ and $O'_{i+1}$, we need two forward traversals to $A_{n,1}$:

$$FT = [S_i/P] + [S_{i+1}/P] + \ldots + [S_n/P].$$

To simplify the cost model, we assume that $O_{i+1}$ and $O'_{i+1}$ have different key values and that they are in different leaf node pages of the path index. To search through the nonleaf nodes of the path index and to read and write the leaf pages for $O_{i+1}$ and $O'_{i+1}$, the number of page accesses needed is:

$$CO = h + 2[XP/P].$$
Therefore, the number of pages accesses for update with the path index is:

\[ PIU = 2 (CO + FT). \]  

(4.10)

**Path Dictionary**

To update the complex attribute \( A_{i+1} \) of \( O_i \) from \( \theta_{i+1} \) to \( \theta'_{i+1} \), the identity index is searched to read and write back the s-expressions corresponding to \( \theta_{i+1} \) and \( \theta'_{i+1} \). To simplify our analysis, we assume that \( \theta_{i+1} \) and \( \theta'_{i+1} \) are in different s-expressions and that they are in different pages. Therefore, the number of page accesses for update is:

\[ PDU = 2 (h_{\text{identity}} + 2 + 2 \lceil SS/P \rceil), \]  

(4.11)

where \( h_{\text{identity}} = \text{height of the identity index} - 1. \)

**Path Dictionary Index**

There are three different cases in which we have to update the path dictionary index:

1. An indexed simple attribute is modified: in this case, the update necessary for the PDI is to update the attribute index involved. Since two index scans are needed, the number of page accesses for update with PDI is:

\[ PDIU = 2 (h_{\text{attr}} + 2 \lceil XP_{A_{i,j}} / P \rceil), \]  

(4.12)

where \( h_{\text{attr}} = \text{height of the attribute index} - 1. \)
2. The complex attribute $A_{i+1}$ of $O_i$ is changed from $\theta_{i+1}$ to $\theta'_{i+1}$, and no attribute in $C_i$ and $C_i$'s ancestor classes are indexed. In this case, the update cost is the same as that of the path dictionary (see Equation 4.11).

3. If one of the attributes in $C_i$ or $C_i$'s ancestor classes is indexed, the attribute index has to be updated too. Therefore, the number of page accesses for update with PDI is:

$$PDIU = 2(h_{identity} + 2 + 2[SS/P]) + 2(h_{attr.} + 2[XP_{attr.}/P]),$$ (4.13)

where $XP_{attr.}$ is the size of a leaf node record in the attribute index.

**Comparison**

In the first case, both the path index and the path dictionary index are required to update their indexes, while the path dictionary mechanism is not required to do so. For the second case, all of the three mechanisms need an update, while the path dictionary and path dictionary index have the same update cost. For the third case, it's only fair to compare the path dictionary and path dictionary index, because the indexed attribute in the path index must be at the leaf class of the path; this is why the path index doesn't support PT queries and it doesn't need an update in this situation. In our comparison, we choose the formula for case 3 to compute the update cost of the path dictionary index, $PDIU$. Since $PDU$ and $PDIU$ are the same for the second case, $PLU$ and $PDU$ are used to compare the update costs between the path index and the other two mechanisms. The difference between $PDIU$ and $PDU$
is the update overhead on the attribute index required for the path dictionary index method.

![Graph showing update cost](image)

Figure 18: Update cost.

One of the most important reasons for extending the path dictionary with the identity index is to improve its update performance. The improvement is shown in Figure 18, which depicts the update cost of the three methods. Initially, the path index has the same update cost as that of the path dictionary and the path dictionary index. However, as $K$ increases, the update cost of the path index dramatically increases, while the update cost for the two path dictionary mechanisms remains relatively low. The difference between $PDU$ and $PDIU$ decreases as $K$ increases,
because, owing to assumptions 1 and 3, there are fewer key values in the indexed attribute and fewer objects in $C_4$ when $K$ is large. Therefore, the size of the attribute index is smaller with large values of $K$, resulting in smaller update overhead on the attribute index.
### 4.5.5 Summary of Formulae for Path Index and Path Dictionary Organizations

In the following, Table 4 summarizes the formulae for path index, path dictionary and path dictionary index.

Table 4: Summary of formulae for path index and path dictionary organizations.

#### Storage Overhead:

<table>
<thead>
<tr>
<th>Organizations</th>
<th>Formulae</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path Index</td>
<td>( PIS = LP + NLP ).</td>
<td>pp. 67</td>
</tr>
<tr>
<td>Path Dictionary</td>
<td>( PDS = FSD + SSP + IIP ).</td>
<td>pp. 69</td>
</tr>
<tr>
<td>Path Dictionary Index</td>
<td>( PDIS = FSD + SSP + IIP + AIP_{index_1} + \ldots + AIP_{index_m} ).</td>
<td>pp. 70</td>
</tr>
</tbody>
</table>

#### Retrieval Cost:

<table>
<thead>
<tr>
<th>Organizations</th>
<th>Formulae</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path Index</td>
<td>( PIR = \begin{cases} h + 1 &amp; \text{if } XP \leq P \ h + \lceil XP/P \rceil &amp; \text{if } XP &gt; P \end{cases} )</td>
<td>pp. 74</td>
</tr>
<tr>
<td>Path Dictionary</td>
<td>( PDR = \lceil N_p S_p/P \rceil + N_{plQ} \cdot (h_{identity} + 1 + \lceil SS/P \rceil) )</td>
<td>pp. 74</td>
</tr>
<tr>
<td>Path Dictionary Index</td>
<td>( PD_{IR} = h_{attr.} + \lceil XP_{attr.}/P \rceil + N_{plQ} \cdot \lceil SS/P \rceil )</td>
<td>pp. 74</td>
</tr>
</tbody>
</table>

#### Range Query Cost:

<table>
<thead>
<tr>
<th>Organizations</th>
<th>Formulae</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path Index</td>
<td>( PIRQ = \begin{cases} h + \lceil NRQ/NREC \rceil &amp; \text{if } \ldots \ h + \lceil NRQ/\lceil XP/P \rceil \rceil &amp; \text{if } \ldots \end{cases} )</td>
<td>pp. 78</td>
</tr>
<tr>
<td>Path Dictionary</td>
<td>( PDRQ = \lceil N_p S_p/P \rceil + N_{plQ} (h + 1 + \lceil SS/P \rceil) )</td>
<td>pp. 79</td>
</tr>
<tr>
<td>Path Dictionary Index</td>
<td>( PD_{IRQ} = \begin{cases} h + \lceil NRQ/NREC \rceil + \ldots \ h + \lceil NRQ/\lceil XP/P \rceil \rceil + \ldots \end{cases} )</td>
<td>pp. 79</td>
</tr>
</tbody>
</table>

#### Update Cost:

<table>
<thead>
<tr>
<th>Organizations</th>
<th>Formulae</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path Index</td>
<td>( PIU = 2(\alpha + FT) ).</td>
<td>pp. 82</td>
</tr>
<tr>
<td>Path Dictionary</td>
<td>( PDU = 2(h_{identity} + 2 + 2\lceil SS/P \rceil) )</td>
<td>pp. 82</td>
</tr>
<tr>
<td>Path Dictionary Index</td>
<td>( PD_{IU} = \frac{2}{2} (h_{identity} + 2 + 2\lceil SS/P \rceil) + 2(h_{attr.} + 2\lceil XP_{attr.}/P \rceil) )</td>
<td>pp. 83</td>
</tr>
</tbody>
</table>
4.6 Direct Links Organization

The idea of the direct link index (DLI) is similar to that of the path dictionary index. Instead of maintaining a path dictionary, we create direct links for classes which are frequently accessed together. In comparison, the path dictionary in PDI encodes the exact aggregation hierarchy, whereas DLI may create links not found in the aggregation hierarchy. The direct link index is very effective for object retrieval, as we will show in our performance analysis. For a nested query, if the direct links between the target and predicate classes are available, the access to objects located in the intermediate classes between target class and predicate classes may be avoided.

Take the the database described in Chapter 2 as an example. Suppose direct links are built between Person and Company. Since Person is an ancestor class of Company, Person is the a-class and Company is the c-class (c stands for child) of the direct links organization. To create the direct links, we first traverse the objects from Person to Company to collect the OIDs of the connected objects from these two classes. In the example below, \((\text{Person}[i], \text{Company}[j])\) represents a link between Person\([i]\) and Company\([j]\). Duplicated links (e.g., Person\([6]\) and Company\([7]\)) mean there are alternative paths between the linked objects. We maintain duplicated links in the list in order to prevent the update problem discussed in the nested index method.

\[
\begin{align*}
\text{Person}[1] & \quad \text{Company}[4] \\
\text{Person}[3] & \quad \text{Company}[2] \\
\text{Person}[5] & \quad \text{Company}[2] \\
\text{Person}[8] & \quad \text{Company}[2] \\
\text{Person}[2] & \quad \text{Company}[7] \\
\text{Person}[6] & \quad \text{Company}[7] \\
\text{Person}[6] & \quad \text{Company}[7] \\
\end{align*}
\]
The direct links provide short cuts between the Person and Company classes. However, nested queries usually are coupled with predicates for associative search. Take the query Q1 "retrieve persons who have GM cars" as an example, we have to search the Company class based on "Name= GM". Tree-structured index and signature file methods [24,37] are two of the most important techniques for associative search. They may be combined with direct links to produce object access mechanisms for nested query processing.

4.6.1 Definition of Direct Links

Given a path $P$ consisting of $C_1C_2...C_n$ in the aggregation hierarchy, the direct links $DL$ on $P$ is defined as follows.

$$DL = \{(\theta_1, \theta_n)|OID(O_1) = \theta_1, OID(O_n) = \theta_n; O_1 \in C_1, O_n \in C_n; O_1 \rightsquigarrow O_n\}.$$

$O_1 \rightsquigarrow O_n$ means that $O_n$ is nested in $O_1$. Since $C_1$ is an ancestor class of $C_n$, we call $C_1$ an a-class and $C_n$ a c-class of the direct links. Each pair of the OIDs in $DL$ is a direct link. The direct links in $DL$ may be sorted by a-class OIDs or by c-class OIDs.

The direct links may be implemented at the system level. We may traverse from the target class to a predicate class and then store the direct links obtained as an extended complex attribute of the target class. Similarly, links obtained by traversing from a predicate class to the target class may be stored as an extended attribute of the predicate class. However, this method doesn't provide a unified structure to support both forward and backward traversals. Further, it is desirable to retain the
organization of the original database. Therefore, we choose a sequential file to store the flat direct links, the pairs of OIDs from a target and a predicate class.

### 4.6.2 Direct Links Indexes

One of our requirements in designing the structure of direct links is to support multiple access methods. In other words, we must be able to couple the direct links structure with arbitrary indexes built on the attributes of the $a$-class or $c$-class of the direct links organization. Thus, the organization of direct links index consists of two levels: the upper level is the attribute indexes, which are tree-structured associative searching mechanisms (e.g., $B^+$-trees), and the lower level is the direct links file, which is accessed through the attribute indexes.

![Diagram of Direct Links Index](image)

**Figure 19:** Direct links index.

Therefore, as shown in Figure 19, the direct links index has a similar architecture as the path dictionary index. An index may be created on one of the attributes from
the \textit{a}-class or \textit{c}-class. The attribute is called the \emph{key attribute} of the index. The \textit{a}-class and the \textit{c}-class of the direct links are the \emph{range classes} of the index.

\textbf{Forward and Backward Indexes}

The \emph{forward index} maps from an attribute in the \textit{a}-class to a list of direct links. Given a list of direct links, \(DL\), and an attribute \(A\) in the \textit{a}-class of \(DL\), a forward index may be defined as follows.

\[ F/I = \{(\kappa, \theta_a, \theta_c) | (\theta_a, \theta_c) \in DL; A(\theta_a) = \kappa\}. \]

The following is an example of the forward index for the path \texttt{Person.Vehicle.Company}.

\texttt{Person.Age} is the key attribute.

\begin{center}
\begin{tabular}{ccc}
24 & \texttt{Person[1]} & \texttt{Company[4]} \\
33 & \texttt{Person[3]} & \texttt{Company[2]} \\
42 & \texttt{Person[5]} & \texttt{Company[2]} \\
50 & \texttt{Person[8]} & \texttt{Company[3]} \\
50 & \texttt{Person[8]} & \texttt{Company[3]} \\
60 & \texttt{Person[6]} & \texttt{Company[7]} \\
60 & \texttt{Person[6]} & \texttt{Company[7]} \\
\end{tabular}
\end{center}

As its name implies, the forward index may be used for forward traversal. Therefore, \textit{PT} queries benefit most from this organization. We use \texttt{Q4: "retrieve the manufacturers of the cars owned by persons at the age of 50."} as an example to illustrate the query processing with the forward index. We first scan the index to locate the direct links corresponding to \texttt{"Person.Age=50"}. Through the direct links collected, the OIDs of the \texttt{Company} objects are returned. Similarly, a query such as \texttt{"retrieve persons at the age of 50"} may be directly answered from the index without accessing any objects in the database.
On the other hand, a backward index is created based on one of the attributes in the c-class. Given a list of direct links, $DL$, and an attribute $A$ of the c-class of $DL$, a backward index may be defined as follows:

$$BI = \{(\kappa, \theta_a, \theta_c) \mid (\theta_a, \theta_c) \in DL; A(\theta_c) = \kappa\}.$$ 

The following is an example of the backward index for the path Person.Vehicle.Company:


The backward index supports backward traversal. It may be used to answer TP queries. For instance, to evaluate the query "retrieve persons who have cars made by Ford", we scan the backward index organization to locate the direct links corresponding to "Company.Name= Ford" and return the Person OIDs in the qualified links.

With the index organization, no objects in the intermediate classes are accessed from the database. However, for queries with multiple predicates (e.g., Q1), we still have to access some of the objects derived from the direct links for further predicate evaluation, unless indexes are available for all of the predicates in the query.

**Clustering Index and Non-clustering Indexes**

To provide fast access to the direct links, the direct links file may be clustered on a given attribute. However, only one attribute can be selected for clustering the direct
links file. We call the attribute a *clustering attribute*. The index organization based on a clustering attribute is called a *clustering index*.

For every cluster in the direct links file, the direct links are sorted by the OIDs in the class containing the clustering attribute. Thus, non-clustering indexes may be further divided into two categories:

- **Semi-clustering index**: the indexed attribute is in the same class with the clustering attribute.
- **Non-clustering index**: the indexed attribute is not in the same class as the clustering attribute.

Take the direct links for `Person.Vehicle.Company` as an example. If the direct links file is clustered on `Company.Name`, then `Company.Location` is a semi-clustering attribute and `Person.Age` is a non-clustering attribute.

After searching a clustering attribute index, the direct links corresponding to the searched attribute value can be sequentially read from the direct links file. For non-clustering attributes, the corresponding direct links are scattered in the direct links file so that random reads are necessary. On the other hand, semi-clustering indexes may have better performance than that of non-clustering indexes, because the corresponding direct links are more likely to be stored together.

**Structures of Attribute Indexes**

The clustering, semi-clustering and non-clustering indexes have similar data structures. We choose the $B^+$-tree for the implementation of the attribute indexes of these
Figure 20 illustrates the data structures used for the attribute indexes. Figure 20(a) shows a nonleaf node page for the clustering and non-clustering attribute indexes. Figure 20(b) is a leaf node record for a clustering index, and Figure 20(c) is a leaf node record for a non-clustering index.

The leaf node record of a clustering index is smaller than that of a non-clustering index. A leaf record keeps the location, specified as a page pointer and a link number, and the number of direct links corresponding to a specific key value. Since the direct links file is clustered by its key values, the collection of direct links may be sequentially read.

For non-clustering indexes, the leaf node records have to keep track of the locations of all corresponding direct links. Due to the way direct links are clustered, a semi-clustering index is likely to have some direct links sequentially stored in the direct links file. To simplify our design, however, we use the same leaf-record structure for both semi-clustering indexes and non-clustering indexes.
4.6.3 Analytical Cost Model

In the direct links index organizations we proposed, there is no structural difference between the implementations of forward and backward indexes. However, due to the clustering of the direct links file, clustering indexes and non-clustering indexes have different storage overheads and retrieval speeds. Although a semi-clustering index has the same storage overhead as a non-clustering index, we expect that it has a different retrieval performance. In this section, we build a forward index for the path $C_1C_2...C_n$ using different implementations, namely the clustering, semi-clustering and non-clustering indexes. We develop analytical cost models for each implementation and compare their performance in terms of storage overhead and retrieval cost.

The Parameters of the Cost Model

Given a path, $C_1C_2...C_n$, the parameters of the cost models for the three index implementations are listed in Table 5. Some of the parameters are used to characterize the classes and their attributes on the path, while others are used to describe the structures of these organizations.

Performance is measured by the number of I/O accesses. Since a page is the basic unit for data transfer between the main memory and the external storage, we use it to estimate the storage overhead and the performance cost. All lengths and sizes used above are in bytes. Table 6 lists the parameter values used in the analysis.

To simplify the cost models, we assume that:

1. There are no partial instantiations, which implies $D_i = N_{i+1}$. 
Table 5: Parameters and symbols for cost models of direct link indexes.

- $C_a$: $a$-class for the direct links, $C_a = C_1$.
- $C_c$: $c$-class for the direct links, $C_c = C_n$.
- $N_i$: the number of objects in class $C_i$, $1 \leq i \leq n$.
- $S_i$: the average size of an object in class $C_i$.
- $A_i$: the complex attribute of $C_i$ used to connect $C_i$ to $C_{i+1}$, $1 \leq i \leq n - 1$.
- $D_i$: the number of distinct values for complex attribute $A_i$.
- $B_{ri}$: the average number of branches from $C_i$ to $C_{i+1}$.
  \[ B_{ri} = \frac{\text{number of OIDs in } A_i}{N_i} \]
- $S_{ri}$: the ratio of shared references from $C_i$ to $C_{i+1}$.
  \[ S_{ri} = \frac{\text{number of OIDs in } A_i}{N_{i+1}} \]
- $A$: the attribute of $C_a$ selected for indexing.
- $U$: the number of distinct values in $A$.
- $q$: the ratio of shared attribute values between objects in class $C_a$ and values for attribute $A$.
  \[ q = \frac{N_a}{U} \]
- $OIDL$: the length of an object identifier.
- $P$: the page size.
- $pp$: the length of a page pointer.
- $ll$: the length of a link number.
- $f$: average fanout from a nonleaf node in the attribute index.
- $kv$: average length of a key value in $A$.
- $kl$: the length of the key length field in the attribute index.
- $cl$: the length of the counter fields (i.e., no. of links).
- $h$: the number of levels of non-leaf nodes in the attribute index.

Table 6: Parameter values for the cost models of direct link indexes.

<table>
<thead>
<tr>
<th>$P$</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>$OIDL$</td>
<td>8</td>
</tr>
<tr>
<td>$pp$</td>
<td>4</td>
</tr>
<tr>
<td>$cl$</td>
<td>4</td>
</tr>
<tr>
<td>$ll$</td>
<td>2</td>
</tr>
<tr>
<td>$kl$</td>
<td>1</td>
</tr>
<tr>
<td>$kv$</td>
<td>8</td>
</tr>
<tr>
<td>$f$</td>
<td>218</td>
</tr>
</tbody>
</table>
2. All key values have the same length.

3. Attribute values are uniformly distributed among the objects of the class containing the attribute.

Storage Overhead

In the following, we develop the formulae of storage overheads for implementing the forward index as a clustered index and a non-clustered index, respectively. Since the direct links organization and nonleaf nodes of the attribute indexes are common to all of the clustered index and non-clustered index implementations, we first develop the formulae for storage overhead of the direct links and nonleaf nodes of the attribute indexes.

Direct Links: In the direct links file, each direct link consists of two OIDs. Therefore, the size of a direct link is \((2 \cdot \text{OIDL})\) bytes. The number of direct links in the file can be determined by:

\[
NDL = N_1 \cdot B_{r_1}B_{r_2}...B_{r_{i-1}} = S_{r_1}S_{r_2}...S_{r_{i-1}} \cdot N_n.
\]

The total number of pages needed for the direct links file is:

\[
SDL = \lceil NDL \cdot (2 \cdot \text{OIDL}) / P \rceil.
\] (4.14)

Nonleaf Nodes: All three implementations have the same structure for nonleaf node records. The number of nonleaf pages can be derived from:

\[
NLP_i = \lceil IO_i / f \rceil + \lceil [IO_i / f] / f \rceil + ... + X,
\]
where $LO_i = \min(U, LP_i)$, $LP_i$ is the number of leaf node pages for implementation $i$, and $X < f$. If $X \neq 1$, $NLP_i$ is incremented by one to account for the root node.

**Clustering Index:** The length of a leaf node record for the clustering index implementation is:

$$XP_{ci} = kl + kv + pp + ll + cl.$$  

Therefore, the number of leaf node pages needed is:

$$LP_{ci} = \lceil U/[P/XP_{ci}] \rceil.$$  

The storage overhead for implementing the forward index as a clustering index is:

$$SCI = LP_{ci} + NLP_{ci} + SDL.$$  \hspace{1cm} (4.15)

**Non-clustering Indexes:** The average number of direct links associated with an object in $C_a$ is:

$$BR = Br_1 Br_2 \ldots Br_{n-1}.$$  

The average number of objects in $C_a$ corresponding to a key value in attribute $A$ is $q$. Thus, the average length of a leaf node record for a non-clustering index is:

$$XP_{ni} = kl + kv + cl + q \cdot BR \cdot (pp + ll).$$  

Therefore, the number of leaf node pages needed is:

$$LP_{ni} = \begin{cases} 
\lceil U/[P/XP_{ni}] \rceil, & \text{if } XP_{ni} \leq P \\
U[XP_{ni}/P], & \text{if } XP_{ni} > P.
\end{cases}$$  

The storage overhead for implementing a forward index as a non-clustering index is:

$$SNI = LP_{ni} + NLP_{ni} + SDL.$$  \hspace{1cm} (4.16)
Comparison: We use the above formulae to compare the storage overheads of a forward index when implemented as a clustering or non-clustering index. In our comparison, we use a path of four classes, $C_1C_2C_3C_4$. An attribute of $C_a$ with sharing ratio $q = 20$ is chosen as the key attribute of the forward index. We also fix the number of objects in $C_a$ to 100000 and the average size of an object to 100 bytes.

$BR$ is the average number of direct links corresponding to an object in $C_a$. It is critical to the storage overhead of the direct links file and the non-clustering implementation. Therefore, we vary $BR$ to observe its impact on storage requirements. From Figure 21, we observe that the storage overhead for both implementations are proportional to $BR$. Since there is little difference between the storage requirements.
of the clustering and non-clustering methods, we conclude that most of the storage overhead is attributable to the direct links file. Fortunately, a single direct links file is shared among many indexes. Thus, the amortized overhead for each index is small.

**Retrieval Cost**

Using the classification introduced in Chapter 2, queries on nested objects can be classified as $TP, PT$ and $MX$. Since we use a forward index for comparison, we will only consider $PT$ queries. Backward index which supports $TP$ queries should have comparable cost and performance as the forward index. We estimate the cost for evaluating $PT$ queries with different implementations of the forward index. The predicate class is $C_a$ and the target class is $C_c$.

**Clustering Index:** The query is answered by first searching through the nonleaf and leaf nodes of the index, and then sequentially reading the direct links from the direct links file. The number of pages accessed for retrieval is:

$$RCI = h + 1 + \left\lceil \frac{q \cdot BR \cdot 2 \cdot OIDL}{P} \right\rceil. \quad (4.17)$$

**Non-clustering Index:** Queries can be answered with non-clustering indexes in the same way as with clustering indexes. However, the number of page accesses to the leaf nodes depends on the number of links corresponding to the key value. Further, since the direct links are not clustered, sequential read can't be used to fetch direct links.
The number of page accesses for randomly accessing \( k \) records from a file of \( n \) records which are divided into \( m \) pages is [53]:

\[
H(k, m, n) = m \left[ 1 - \prod_{i=1}^{k} \frac{n - (n/m) - i + 1}{n - i + 1} \right].
\]

The average number of links accessed from the direct links file is \( q \cdot BR \). The number of direct links in the file is \( NDL \) and the number of pages for the file is \( SDL \). Applying the above formula, we obtain the number of page accesses to the direct links file:

\[
RDL_{ni} = H(q \cdot BR, SDL, NDL).
\]

As a result, the total number of page access for retrieval using a non-clustering index is:

\[
RNI = \begin{cases} 
  h + 1 + RDL_{ni} & \text{if } X_{ni} \leq P \\
  h + \lceil X_{ni}/P \rceil + RDL_{ni} & \text{if } X_{ni} > P,
\end{cases}
\]

where \( X_{ni} \) is the size of a leaf node record in a non-clustering index.

**Semi-clustering Index:** Although query processing is similar in the semi-clustering and non-clustering implementations, the number of page accesses to the direct links file is lower for the non-clustering index method because the links in each cluster are sorted by the OIDs in \( C_a \). Therefore, the number of direct links accessed is \( q \cdot BR/K \), where \( 1 \leq K \leq BR \). \( K \) is the average number of consecutive direct links with the same OIDs in \( C_a \). Each random access to the direct links file will read \((K \cdot 2 \cdot OIDL)\) bytes sequentially. Thus, the number of page accesses to the direct links file is:

\[
RDL_{si} = H\left(\frac{q \cdot BR}{K}, SDL, NDL\right) \cdot \left\lceil (K \cdot 2 \cdot OIDL)/P \right\rceil.
\]
Thus, the total number of page accesses for retrieval using the semi-clustering index is:

\[
RSI = \begin{cases} 
  h + 1 + RDL_{si} & \text{if } XP_{si} \leq P \\
  h + \lceil XP_{si}/P \rceil + RDL_{si} & \text{if } XP_{si} > P,
\end{cases}
\]  (4.19)

where \( XP_{si} = XP_{ni} \) is the size of a leaf node record in the semi-clustering index.

**Traditional Forward Traversal:** To answer a PT query with forward traversal, the number of pages accesses is:

\[
RFT = \lceil N_a \cdot S_a/P \rceil + q \cdot (B_{r1} + B_{r1}B_{r2} + ... + B_{r1}B_{r2}...B_{rn-1}) .
\]  (4.20)

![Figure 22: Retrieval cost.](image-url)
Comparison: In the comparison, we fix $K$ to 2 when we estimate the retrieval cost of the semi-clustering method. From Figure 22, the clustering index method has the best retrieval performance, which is consistent with our expectation. The retrieval cost of semi-clustering index is between that of the clustering and non-clustering indexes. If we increase the value of $K$ for the semi-clustering index, its retrieval cost will shift toward the performance of the clustering index method.

Summary of Formulae for Direct Link Organizations

In the following, Table 7 summarizes the formulae we developed for direct link organizations.

Table 7: Summary of formulae for direct link organizations.

<table>
<thead>
<tr>
<th>Storage Overhead:</th>
<th>Formulae</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Links</td>
<td>$SDL = \lfloor NDL \cdot (2 \cdot OIDL) / P \rfloor$</td>
<td>pp. 96</td>
</tr>
<tr>
<td>Clustering Index</td>
<td>$SCI = LP_{ci} + NLP_{ci} + SDL$</td>
<td>pp. 97</td>
</tr>
<tr>
<td>Non-clustering Index</td>
<td>$SNI = LP_{ni} + NLP_{ni} + SDL$</td>
<td>pp. 98</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Retrieval Cost:</th>
<th>Formulae</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clustering Index</td>
<td>$PCI = h + 1 + [q \cdot BR \cdot 2 \cdot OIDL / P]$</td>
<td>pp. 99</td>
</tr>
<tr>
<td>Non-clustering Index</td>
<td>$RNI = \begin{cases} h + 1 + RDL_{ni} &amp; \text{if } \ldots \ h + [XP_{ni} / P] + RDL_{ni} &amp; \text{if } \ldots \end{cases}$</td>
<td>pp. 100</td>
</tr>
<tr>
<td>Semi-clustering Index</td>
<td>$RSI = \begin{cases} h + 1 + RDL_{si} &amp; \text{if } \ldots \ h + [XP_{si} / P] + RDL_{si} &amp; \text{if } \ldots \end{cases}$</td>
<td>pp. 101</td>
</tr>
<tr>
<td>Forward Traversal</td>
<td>$RFT = \lfloor Na \cdot Sa / P \rfloor + q \cdot (Br_{1} + Br_{1}Br_{2} + \ldots + Br_{1}Br_{2}\ldots Br_{n-1})$</td>
<td>pp. 101</td>
</tr>
</tbody>
</table>
4.7 Summary

In this chapter, we have shown that the path information embedded among objects can be exploited to significantly improve the performance of nested object queries in object-oriented databases. We present two indexing organizations for nested object query processing. A novel feature of these two organizations is that indexes on arbitrary attributes can be built with only a small amount of extra storage overhead.

For the path dictionary index, we develop cost models for the storage overhead and retrieval and update costs, and compare the costs to the path index method. For most of the comparisons, we vary the average ratio of shared references and shared key values to observe its impact on the performance and overhead of the path index, path dictionary, and path dictionary index. As a result, we conclude that the path dictionary index is superior to the path index.

Direct links are effective in providing short cuts for object traversal. Based on direct links, a new class of indexing mechanisms, forward index and backward index, for both forward and backward traversals is proposed. Because of the clustering of direct links, a forward or backward index can be realized in one of the following ways: clustering index, semi-clustering index, or non-clustering index. Based on this classification, we develop cost models for the estimation and comparison of their storage overheads and retrieval costs.

Our analysis shows that the clustering index implementation has the best performance in storage overhead and retrieval speed. Semi-clustering index has the same storage overhead as the non-clustering index, but its retrieval speed is faster. Al-
though non-clustering index is not as good as the other two implementations, it is still much better than the traditional traversal approaches.
CHAPTER V

Information Broadcast in Mobile Computing Environments

5.1 Introduction

Rapid advances on wireless data networks and personal computing open up new services and activities for mobile users. Various commercial and experimental mobile computers have appeared recently [7,52]. It is envisioned that mobile computing techniques will bring a revolution for the future computing systems. Within a wireless communication environment, the models of information access are not limited to those used for the traditional wired media. New information access approaches such as broadcast and filtering have to be considered.

In this chapter, we first introduce the architecture of mobile computing systems. Then, we discuss the broadcast approach of information dissemination and its performance evaluation criteria, i.e., access time and tuning time. Finally, we further discuss the issues for caching signatures with the mobile computers.

5.2 Mobile Computing System Architecture

A mobile computing system consists of a set of mobile computers and stationary hosts connected by a fixed network [21]. The mobile computers communicate with other
computers through wireless communication, while the stationary hosts communicate with each other through the wired network. In order to support the wireless communication for mobile computers, the geographical area covered by the system is divided into cells. The wireless communication in a cell is supported by the mobile support station (MSS), which is one of the stationary computers on the fixed network. The MSS controls the wireless communication channels in a cell. In addition to supporting communication, the MSS provides various services to the mobile users. As a result, the MSS may be considered as a database server which is responsible for providing and disseminating data to its mobile clients in the cell. Figure 23 shows the architecture for supporting mobile computing and illustrates the terminology such as mobile computers, the MSSs, and cells.

The application of information broadcast is numerous. An example is mobile shopping, where product information is broadcast continuously and users with mobile computers can specify the products they are interested in, browse through the
information provided (e.g., sample pictures, prices, and so on), and order merchandise with a few key strokes. This example can be extended to shopping malls, where information such as product categories, locations of stores, special sales can be provided to customers on the air. Experimental shopping assistant systems based on these ideas have been proposed and developed [4].

We may consider two modes of operations in information broadcast applications. For mobile computers with transmit capability, the mobile computers will send the user queries to the central server, which collects the queries over a period of time and broadcasts the requested information on the channel. The mobile computers are responsible for identifying the information they need from the channel. For mobile computers without transmit capability, the central server will broadcast all of the information available on the channel in a certain order, the mobile computers listen to the channel and select the information requested by the users. It is noted in either case that information from the central server is broadcast on a single channel because allocating a dedicated channel to each individual mobile computer is too expensive and that the mobile computers are required to filter out unwanted information from the channel so as to reduce the amount of information presented to the users and to reduce battery consumption by the mobile computers.

5.3 Power Conservation and Information Broadcast

A major problem with mobile computers is their power supplies. Batteries are the main power source in most mobile computers. In order to make mobile computers
portable, small batteries, such as AA or AAA batteries, are likely to be used [5]. However, these batteries have small capacity and need recharging or replacement after a short period of usage. Although processors and memories consuming less power have been developed (e.g., the Hobbit chip from AT&T and energy efficient chip designs at Berkeley [12]), new generations of faster chips with high clock frequency will continue to demand more and more power. Therefore, power conservation is an important issue for applications in a mobile computing environment.

There are two factors affecting power consumption in mobile computers: (1) mobile computers can be switched between active mode (full power) and doze mode [18], and (2) receiving messages consumes less energy than sending messages. Methods have been proposed for energy efficient query optimization [3] and information broadcast [22,23].

In the wireless environment, broadcast is an attractive method of disseminating information to mobile users because base stations are equipped with powerful communication equipment but mobile computers, restricted by cost, portability and power, can afford little or no transmission capability. Moreover, broadcast can scale up to an arbitrary number of users. In contrast to accessing traditional storage media, the performance of accessing information on air does not degrade as the number of users increases.

In broadcast, the base station sends out a series of information frames. (See Figure 24.) An information frame is a logical unit of information broadcast on the air and may consist of multimedia information, including text, image, audio/video and
other related data. Frames may vary in size; they consist of packets which are the physical units of the broadcast. A frame contains a header (not shown in the figure) for synchronization as well as meta-information indicating the type and length of the frame. At the receiving end, users are allowed to specify conditions on the frames they are interested in. The mobile computers will only present to the users frames matching the conditions. Since the information frames are periodically broadcast, a complete broadcast of the information frames is called a broadcast cycle. From the user's viewpoint, the broadcast information is perceived as a stream of frames flowing along the time axis. Logically, there is no specific start and end frames for a broadcast cycle; a broadcast cycle starts with any frame and ends when the frame appears again. In a broadcast cycle, some important information frames may be replicated (i.e., frames with the same contents but treated as different frames). Information frames may be inserted, deleted, and modified. The updates will be reflected in the subsequent broadcast cycles.

The duration that a mobile computer must stay in active mode to answer a query is called the tune-in time. The power consumption of a mobile computer used for information broadcast and filtering services is proportional to the tune-in time. Access
time is the time required to collect all qualified frames. Without any access aid, both the tune-in time and access time are equal to the length of the broadcast cycle, because it is necessary to scan through all of the frames in a broadcast cycle to pick up the qualified frames. This is very inefficient in power consumption, because typically only a few frames in a cycle satisfy the user request.

Access methods can be developed so that the mobile computer can be turned off when the frame being broadcast is not qualified. By switching between active and doze modes, power consumption is reduced. In order to tell which frames would qualify ahead of time, auxiliary information about the contents of the frames must be added. Due to the limited number of broadcast channels available, we assume that only one channel is used for both primary and auxiliary information. With only one channel, the auxiliary information will increase the length of a broadcast cycle and thus increase access time. However, it will reduce tune-in time, because it allows the mobile computers to avoid tuning into unwanted information frames. Thus, we must tradeoff between access time and tune-in time when we consider what auxiliary information is to be used and how it is going to be organized.

Two approaches, namely, hashing and indexing [22,23], have been proposed in the literature for encoding auxiliary access information for wireless broadcast information services. Organizations and algorithms for disseminating and retrieving data on air were proposed. Tradeoffs between power consumption and access time were also considered. However, indexes based on one single key were assumed in these studies [22,23]. Therefore, they won’t support general queries involving various attributes of
the information frame.

A major difference between traditional disk-based indexing techniques and indexing broadcast data is that disk-based indexing allows random access to data whereas data on a broadcast channel must be accessed sequentially. This property significantly changes the cost factors of indexing techniques. For instance, tree structures are fast on disks because they allow random access from node to node, thus bypassing nodes containing irrelevant data. However, on a broadcast channel, tree structures lose this advantage because it takes time for the mobile computer to skip the irrelevant data.

Signature file techniques are known to be slow compared to tree structures. However, in a broadcast channel, they become very attractive as tree structures lose their advantage in speed. On the other hand, the simplicity of the signature file makes it highly suitable for realtime information filtering under stringent processor speed and memory size. In Chapter 6, we focus on the application of the signature file technique to indexing broadcast information.

5.4 Signature Caching Policies

Previous studies [2,6] have discussed cache management policies and invalidation strategies for information dissemination based on broadcast. In [6], the effect of disconnections on three cache invalidation approaches are discussed, while [2] proposes to use wireless channels as broadcast disks and discuss the associated prefetch and caching strategies. In our study, we consider signature caching schemes for information broadcast and filtering services.
To reduce battery power consumption for mobile computers, we have to reduce the amount of information downloaded from the communication channel. Caching of frequently accessed information in the mobile computers can reduce tune-in time considerably, and thus power consumption, because information can be fetched from the cache without tuning into the communication channel. Due to physical constraints, mobile computers usually have relatively small memory. Thus, caching large chunks of multi-media information frames is infeasible. Compared to the information frames, signatures need a rather small storage overhead and they contain critical information to support various kinds of queries. Thus, they are very suitable as the caching entity.

In this section, we address the issues of caching policies for maintaining signatures in the main memory of the mobile computers to further reduce tune-in time.

A two-level signature scheme is used in our discussion. There are two factors affecting signature maintenance in cache:

- **Invalidation notice**: An invalidation notice indicates which cached signatures are stale.

  In order to support information filtering, the signatures in the cache have to be accurate. Therefore, the information server has to provide invalidation information to mobile computers. If some information frames and their corresponding signatures are updated, the information server will indicate the changes in the invalidation notice embedded in the broadcast.

- **Refresh strategy**: The refresh strategy determines which signatures to maintain in the cache. Since the frequent update of signatures may require the mobile
computers to pay extra attention to the broadcast channel and thus cause more
power consumption, the mobile users may choose to actively or passively refresh
the signatures in cache. The active approach maintains all of the signatures
in the cache, while the passive approach only keeps the previously accessed
signatures in the cache.

Based on the above two factors, we discuss four signature caching policies in
Chapter 7. Analytical cost models and performance evaluation are also presented.

5.5 Related Work

Several organizations for indexing and filtering broadcast data have been proposed in
the literature [22,23,40]. These papers use indexing and hashing techniques to provide
auxiliary information, which is interleaved with the data items for broadcasting. The
main criterion of the performance used in the papers is the tune-in time. The main
issue is to see, with certain access time delay, how much tune-in time may be saved.

5.5.1 Hashing Techniques

The problem of using wireless data broadcast as a way of disseminating informa-
tion to a massive number of battery powered palmtops was discussed in [22], where
two hashing schemes and one flexible indexing method for organizing and accessing
broadcast data were discussed.

The hash-based schemes embedded control information with data buckets, so there
are no extra buckets allocated for the control information. The data buckets contain-
ing hashing function and shift pointers to the beginning of the logical buckets are
called directory buckets. The first hashing scheme puts the directory buckets at the beginning of a broadcast. Missing the directory will delay the filtering process to the next broadcast. The second hashing scheme remedies that problem by modifying the hashing function to interleave control buckets with the other data buckets. Given the same broadcast file and the same search key, the probability of missing the directory bucket for the second scheme is much smaller than that for the first scheme. The cost model for access time has been derived for the hashing schemes. However, the tune-in time is not analyzed. There is a table of data for access time and tune-in time. However, it seems that the initial probe time is not considered in the calculation of the tune-in time.

The flexible indexing method divides the sorted data records into several segments and indexes. The term 'flexible' refers to the size of segments which may be adjusted to get good access and tune-in time. The first bucket of each segment will contain the control index, which consists of binary control index and local index. The binary control index is a partial binary tree which indexes the upcoming data segments in the broadcast cycle. The local index is a linear list which indexes the data buckets in the local segment. The cost models for the tune-in time and access time are derived for the flexible index.

On selecting between the hashing scheme and the indexing method, the author suggested that hashing schemes should be used when the tune-in time requirements are not rigid and the key size is relatively large compared to the record size. Otherwise, indexing method should be used.
5.5.2 Indexing Techniques

Two methods, \((1,m)\) indexing and distributed indexing, for organizing and accessing broadcast data were described in [23]. The paper assumes that the queries are based on the primary key. The data frames are sorted by the primary key. Also, the paper assumes that the size of the data is small enough to fit into a frame.

In the \((1,m)\) indexing method, a tree-structured index, e.g., \(B^+\)-Tree, is created for the data frames in the broadcast cycle. The data frames are divided into \(m\) segments. A replication of the index, called the index segment, is created for each data segment. After accessing the index, the mobile computer may enter doze mode and wake up when the data frame arrives.

The distributed indexing method is based on the observation that there is no need to replicate the entire index between successive data segments. For each data segment, there is an index segment which consists of two parts: the replicated part indexes different data segments and the non-replicated part indexes the data buckets in a data segment.

The cost models for the tune-in time and the access time are derived for both \((1,m)\) indexing and distributed indexing method.

5.5.3 Caching Techniques

Acharya et al. [2] proposed broadcast disks for mobile clients. The broadcast disk superimposes multiple information streams into a super-stream for broadcasting on one single channel. The idea is to interleave the information frames of different streams at
some specified frequencies. The information frames from the more important streams are broadcast more often than those from less important streams. Thus, the frequencies of various streams may be adjusted to reflect the user demands.

In addition to showing how to superimpose information streams into a broadcast channel, the paper discussed caching techniques for broadcast disks. Due to the serial and shared nature of information broadcast, two new caching policies, PIX and LIX, were proposed. Instead of caching the hottest pages, the policies are based on the ratio of local access probability to broadcast frequency. In other words, if the hottest frames may easily be obtained on the air, they need not be cached. The best candidates for caching are those accessed frequently but difficult to obtain. PIX is based on the probability of frame accesses in mobile clients and broadcast frequency of information frames. PIX is not a practical policy for implementation, because the base station requires perfect knowledge of the probability of frame accesses. On the other hand, LIX is implementable, because it only taking into account of the broadcast frequencies. In [55], the same authors also discussed the prefetch strategies of caching for broadcast disks. However, the results are not conclusive.

Three different cache invalidation strategies, broadcasting timestamps, amnesic terminals, and signatures, are proposed in [6]. The timestamp method periodically broadcasts the latest changes to the information frames. The mobile computers listen to the report and update their caches. The amnesic terminal method periodically broadcasts the identifier of items which are changed since the last broadcast of the invalidation report. Finally, in the signature method, a set of combined signatures
is generated from the information frames and broadcast periodically. The mobile computer caches, along with the individual frames of interest, all of the combined signatures of subsets that include items of interest to the user. By comparing the new set of combined signatures with the cached ones, changes on the items in the cache may be identified.

The authors studied the impact of disconnection time on these strategies and concluded that the signature approach is effective for long sleepers and that the timestamp method is the best for the cases when queries are much more frequent than updates. The amnesic terminal method is the best for clients which hardly disconnect.
CHAPTER VI

Signature Techniques for Wireless Information Broadcast

6.1 Introduction

As we introduced in Section 3.2, signature methods have been used in numerous database applications. A signature is an abstraction of the information stored in a record. By examining the signature alone, we can estimate whether the record contains the desired information. Naturally, the signature technique is very suitable for filtering information frames in a wireless broadcasting environment.

In a mobile environment, the signature technique offers the following advantages for information filtering:

- Signature techniques may be generally applied to various types of information media.

- Signature techniques are particular good for multi-attribute retrieval, which is necessary for specifying precise filtering conditions.

- Signatures are very easy to generate and search; thus, they are suitable for mobile computers where realtime searching with limited buffer space is required.
• A signature is very short compared to an information frame; therefore, the access time won't be increased drastically.

• A signature file is basically a sequential file structure. This makes it easy to "linearize" and "distribute" the signature file for broadcast and then scanning by a mobile computer, whereas a tree based access structure will lose the speed advantage because random access cannot be done on a broadcast channel.

In this chapter, we present three signature schemes, simple signature, integrated signature and multilevel signature, for information broadcast and filtering and develop the cost models for their performance evaluation in terms of access time and tune-in time. Moreover, we present four caching policies for the two-level signature scheme and provide performance analysis based on tune-in time.

6.2 Information Broadcast Using the Signature Techniques

In the wireless broadcast environment, signatures are constructed from the information frames and broadcast together with the information frames. The signatures may be broadcast as a group before the information frames or interleaved with the corresponding frames. Figures 25 and 26 illustrate these two approaches.

The period of time from the moment a user tunes in until the first signature is received is called the initial probe time, while the period of time for CPU to change from the doze mode to active mode, or vice versa, is called the setup time. The non-interleaved method simplifies the software for broadcasting at the base stations and minimizes the number of active/doze mode switches at the mobile clients, so it has a
smaller setup time than an interleaved approach. However, since the user may start monitoring the broadcast channels at any moment, missing the signature segment means the user has to wait until the next broadcast cycle to access the signatures. The non-interleaved method results in a longer initial probe time, and thus a longer access time delay, than the interleaved approaches. Moreover, the setup time of a CPU for a typical mobile client, e.g., Piranha chip from AT&T, is less than 20 milliseconds. The setup time advantage of non-interleaved signature method over interleaved methods may not justify its long delay on access time. Thus, we only consider interleaved approaches for the signature schemes discussed in this thesis. Note that during the initial probe time, the user may choose to switch to doze mode until a signature is encountered or to remain in active mode to scan for qualified information frames without the help of signatures. The former will save energy, while the latter may return qualified information frames earlier. Since the focus of our
study is on energy saving, we will assume that the mobile client stays in doze mode during the initial probe time.

Different schemes may be used to organize signatures and information for broadcasting. In this thesis, we discuss three signature methods based on interleaving:

- Simple signature: a signature is constructed and broadcast for each information frame.

- Integrated signature: a signature is constructed and broadcast for a group of information frames.

- Multilevel signature: multiple levels of signatures are constructed and broadcast. The higher levels of signatures are integrated signatures and the lowest level of signatures are simple signatures.

6.2.1 Simple Signature Scheme

The most intuitive approach for interleaving signatures with information frames is to construct a simple signature for each information frame. The signature frame is broadcast before the corresponding information frame (see Figure 26).

When a mobile user wants to retrieve information from the broadcast channel, she/he specifies a query on a mobile client. A query signature $S_Q$ is generated based on the specified query. Then the mobile client tunes into the channel and uses $S_Q$ to compare with the frame signatures received. When a match is found, the corresponding information frame is received by the mobile client for further checking in order to eliminate false drops. If the frame is not a false drop, it will be retained
in the result set. When a received simple signature does not match with the query signature, the mobile client will fetch frame size from the header of the information frame and switch into doze mode until the next signature frame arrives. If most of the simple signatures don't match with the query signature, the mobile client will stay in doze mode for the most part of a broadcast cycle, thus saving a lot of energy.

In the following, we estimate the access time and tune-in time of the simple signature scheme. The setup time for switching back and forth between active and doze modes are very small comparing to the tune-in time saved. Thus, setup time is not taken into consideration for tune-in time. In this scheme, the average initial probe time is half of the average size of an information frame and its signature frame. The access time and tune-in time, however, are dependent on the positions of the initial probe:

**position A:** If the initial probe falls in the middle of a signature frame (point A in Fig. 26), the mobile client has to stay active for the rest of the signature frame in order to detect the beginning of a new frame. Then, the mobile client switches to doze mode. The initial probe time, access time and tune-in time are as follows.

**Initial probe time** = the partial signature scanned + the length of the first information frame.

**Access time** = initial probe time + a broadcast cycle.
**Tune-in time** = length of the partially scanned signature + every signature in a broadcast cycle + false drop and true match information frames in the cycle.

**position B:** If the mobile client initially tunes into the middle of an information frame (point B in Fig. 26), it will stay active for the rest of the frame so that it can detect the beginning of the next signature.

**Initial probe time** = the partial information frame.

**Access time** = initial probe time + a broadcast cycle.

**Tune-in time** = initial probe time + every signature in a broadcast cycle + false drop and true match frames in the cycle.

### 6.2.2 Integrated Signature Scheme

A generalization of the simple signature scheme is to generate a signature, called an *integrated signature*, for a group of one or more information frames, called a *frame group*. The integrated signature is broadcast before the frame group. Figure 27 shows the arrangement of the signatures and frame groups. A signature may index any number of information frames. As shown in the figure, the first signature indexes two information frames while the next signature indexes three information frames. In this scheme, the header of an information frame has to provide information such as the period of time to pass by until the next signature frame arrives.

The filtering procedure of the integrated scheme is very similar to the simple signature scheme. When an integrated signature does not match with the query
signature, the information frames it indexes can be skipped. Since an integrated signature indexes a large number of information frames, the mobile client may stay in doze mode for a long period of time. When a signature match does occur, all of the information frames associated with the signature have to be checked for false drop elimination. Some of the information frames in the true match group may not be qualified for the query, thus resulting in unnecessary power consumption.

An unmatched integrated signature allows the mobile client to stay in doze mode for a longer period of time, thus avoiding frequent switching between modes. However, squeezing more information from multiple frames into a signature will increase the probability of false drops. Therefore, we have to properly adjust the size of signatures or reduce the number of bit strings superimposed into the integrated signatures in order to maintain the filtering capability. Grouping similar information frames (those having the same values for most of the indexed attributes) together to generate the integrated signature is virtually the same as reducing the number of bit strings superimposed. Thus the adjustment will produce more concentrated hits and reduce false drops. This method is good when the order of the information frames is not important.

Figure 27: Integrated signature scheme.
The average initial probe time for this scheme is half of the average size of the information frames grouped together and their integrated signature. Thus, the average initial probe time is longer than the simple signature scheme. As before, we assume the mobile client stays in doze mode during the initial probe time. As in the simple scheme, the initial probe time, access time and tune-in time are dependent on the position of the initial probe.

**position A:** If the initial probe falls in the middle of a signature frame (point A in Fig. 27), the mobile client has to stay active for the rest of the signature frame and then it switches to doze mode.

- **Initial probe time** = length of the partial integrated signature scanned + length of the first group of the information frames.
- **Access time** = initial probe time + a broadcast cycle.
- **Tune-in time** = length of the partial integrated signature scanned + every integrated signature in a cycle + false drop and true match frame groups.

Note that every information frame in the true match group has to be compared with the query, even though only one qualified frame within the group may exist.

**position B:** If the mobile client initially tunes into the middle of an information frame (point B in Fig. 27), it will stay active for the rest of the frame since this is the only way to reach the next frame. If the next frame is an information frame, it will switch to doze mode until the next signature frame arrives.
6.2.3 Multi-level Signature Scheme

The multi-level scheme is a combination of the simple signature and integrated signature schemes [34]. It consists of multiple levels of signatures. Signatures at the higher levels are integrated signatures and those at the lowest level are simple signatures. Thus, we assume that the header of an information frame provides information such as its frame size and the period of time to pass by until the next integrated signature frame arrives. Figure 28 illustrates a 2-level signature scheme. The white signatures in the figure are integrated signatures. The black signatures are simple signatures for the corresponding information frames. An integrated signature indexes all of the information frames between itself and the next integrated signature of the same or at a higher level. (In the figure, an integrated signature indexes two information
frames.) The integrated and simple signatures may use different frame sizes in order to minimize the tune-in time. To simplify the software on base station and mobile clients, however, the signatures may be set to the same size. To reduce the false drop probability, the hashing functions used in generating the integrated signatures and simple signatures are different.

To answer a query, a query signature is generated for each level of signatures. After tuning into the broadcast channel, the corresponding query signatures are used to compare with different levels of signatures. If a signature fails in the comparison, the mobile client switches to doze mode until a signature at the same or higher levels arrives. Otherwise, the mobile client stays in active mode and continues the filtering process. Note that the headers of information frames contains information which may specify the arrival time of the next simple signature and integrated signature. Therefore, the mobile clients may safely switch to doze mode and wake up in time.

Take the 2-level signatures in Figure 28 as an example. Query signatures $S_Q$ and $S_Q'$ are constructed for the integrated and simple signature levels, respectively. When an integrated signature is received, $S_Q$ is used to match with the signature.

- If the match fails, the mobile client will go into doze mode until the next integrated signature arrives, i.e., skip the whole frame group.

- If the match is successful, $S_Q'$ is used to match with each of the simple signatures in the frame group.

  - If $S_Q'$ and a simple signature match, the corresponding information frame is received for false drop elimination.
- if not, the mobile client may go into doze mode until the next simple signature arrives.

Compared to the other schemes, the multi-level scheme may achieve better tune-in time. The initial probe time, access time and tune-in time are dependent on the position of the initial probe.

**position A:** If the initial probe falls on an integrated signature frame (point A in Fig. 28), the mobile client has to be active for the rest of the integrated signature frame. The filtering process will start afterwards.

**Initial probe time** = the partial integrated signature scanned.

**Access time** = initial probe time + a broadcast cycle - the initial integrated signature.

**Tune-in time** = initial probe time + every simple signature in the first frame group + all but the initial integrated signature in a cycle + simple signatures following the qualified integrated signatures + false drop and true match frames associated with the qualified simple signatures.

**position B:** If the initial probe falls on a simple signature frame (point B in Fig. 28), the mobile client has to be active for the rest of the signature frame and then it goes into doze mode as assumed before. The filtering process starts after the next signature arrives.

**Initial probe time** = the partial simple signature frame scanned and its associated information frame.
Access time = initial probe time + a complete broadcast cycle.

Tune-in time = the partial simple signature scanned + every simple signature before the first integrated signature arrives + all of the integrated signature in a cycle + simple signatures associated with the matched integrated signatures + (if the last integrated signature matches) the simple signatures before the initial probe + false drop and true match information frames associated with the matched simple signatures.

position C: If the mobile client initially tunes into the middle of an information frame (point C in Fig. 28), it will stay active for the rest of the frame. Then, filtering starts.

Initial probe time = the partial information frame scanned.

Access time = initial probe time + a broadcast cycle.

Tune-in time = the partial information frame scanned + every simple signature before the first integrated signature + all of the integrated signatures in a cycle + simple signatures associated with the qualified integrated signatures + (if the last integrated signature matches) the simple signatures before the initial probe + false drop and true match information frames associated with the matched simple signatures.

6.2.4 Performance Analysis

There are several factors affecting the tune-in time and access time of the signature schemes. For example, we must consider the number and the size of the signatures,
the filtering capability of the signatures, the false drop probability of the signatures, and the initial probe time. A performance evaluation has to take these factors into account. The filtering capability and false drop probability may be controlled by the size of the signatures. On the other hand, the initial probe time is related to the number of signatures interleaved with the information frames, and the access time and tune-in time are dependent on the number, size and false drop probability of the signatures.

Frame is the the logical unit of information in broadcasting, while packet is the physical unit of information in broadcasting. Therefore, in our analysis, performance is estimated in terms of the number of packets.

Table 8 lists the parameters and symbols used in our development of cost models for simple, integrated and multi-level signature schemes. In the following, we use frame signature to refer to the signatures for information frames, i.e., simple or signature frames.

**False Drop Probability**

The false drop probability is an important factor for the estimation of access time and tune-in time. The false drop probability $P_f$ is generally defined as:

$$P_f = \frac{A_f}{A - A_t}.$$  

where $A$ is the number of information frames in a broadcast cycle, $A_f$ is the number of information frames received due to false drops, and $A_t$ is the number of information frames received due to true matches.
Table 8: Parameters and symbols for cost models of simple, integrated and multi-level signature schemes.

- $A$: number of information frames in a broadcast cycle.
- $A_f$: number of information frames received due to false drops.
- $A_t$: number of information frames received due to true matches.
- $I$: number of integrated signatures in a broadcast cycle.
- $I_f$: number of integrated signatures matched as false drops.
- $I_t$: number of integrated signatures truly matched.
- $P_f$: false drop probability.
- $P_t$: false drop probability for simple signatures.
- $P_{it}$: false drop probability for integrated signatures.
- $P_s$: selectivity of a query.
- $P(x, y)$: probability that a particular set of $x$ bits are set to 1’s in a frame signature which is superimposed from $y$ attribute bit strings.
- $k$: the number of information frames indexed by an integrated signature.
- $l$: locality of true matches (average number of true matches in a frame group).
- $m$: length of a signature in bits.
- $n$: the average number of packets in an information frame.
- $p$: the number of bits in a packet.
- $r$: the number of packets in a signature ($r = \lceil m/p \rceil$).
- $s$: the number of bit strings which are superimposed into a signature.
- $w_b$: number of 1’s in a bit string generated from hashing.
- $w_f$: average number of 1’s in a frame signature ($\bar{w}_f = m - w_f$).
- $\bar{w}_f$: average number of 0’s in a frame signature.

To apply signature techniques to information filtering in mobile computing environments, we have to compare the query signatures with frame signatures of every information frame in order to screen out unqualified information frames. A false drop occurs when each of the bits in the frame signature corresponding to the 1’s in the query signature is set to 1. Let $\alpha_i$ and $\beta_i$ be the $i$-th bit of the query signature and a frame signature, respectively. A false drop occurs when the following condition holds:
if \( \alpha_i = 1 \) then \( \beta_i = 1, \ 1 \leq i \leq m \)

Therefore, a false drop occurs when the following condition holds:

if \( \beta_i = 0 \) then \( \alpha_i = 0, \ 1 \leq i \leq m \)

Let \( \bar{w}_f \) be the average number of 0's in a frame signature, i.e., average number of bits in a frame signature set to 0. Therefore, the false drop probability for signature techniques is the probability that at least \( \bar{w}_f \) bits in the query signature are set to 0. For a query with single key value, the false drop probability for signature techniques is as follows:

\[
P_f = \text{Probability}[\alpha_1 = 0 \land \alpha_2 = 0 \land \cdots \land \alpha_{\bar{w}_f} = 0]
\]

where \( \bar{w}_f \) is the average number of 0's in a frame signature.

In the following, we give two lemmas for the average number of 0's in a signature and the false drop probability for the signature technique. The derivations of the lemmas are included in appendix of the thesis.

**LEMMA 6.2.4.1** Given the length of a signature in bits, \( m \), the number of bit strings superimposed into the signature, \( s \), and the number of 1's set in the bit strings, \( w_b \), the average number of 0's in the signature is:

\[
\bar{w}_f \approx me^{-w_b s/m}.
\]

Next, we derive \( P_f \), assuming an unsuccessful search of a single value query, for the signature techniques. Multiple value queries which are Boolean combinations of
single value queries may be derived similarly. Based on $P_f$, the estimation for the access time and tune-in time can be derived.

**Lemma 6.2.4.2** Given the length of a signature in bits, $m$, the number of bit strings superimposed into the signature, $s$, and the number of 1's set in the bit strings, $w_b$, the false drop probability for the signature is:

$$P_f = \left(1 - \frac{\bar{w}_f}{m}\right)^{w_b}$$

where and $\bar{w}_f$ is the average number of 0's in the signature.

Based on Lemma 6.2.4.1 and Lemma 6.2.4.2, the false drop probability for a signature is derived as follows:

$$P_f \approx (1 - e^{-w_b s/m})^{w_b}.$$  

The above formula is optimal when [49]:

$$w_b = w_{opt} = m \cdot ln2/s.$$  

Consequently, the optimal false drop probability is:

$$P_f \approx 0.5^{w_{opt}}.$$  

**Cost Models**

In this section, we develop the cost models for the initial probe time, access time and tune-in time for the three signature schemes we described. We use the number of packets as the unit for time estimation. To simplify our discussion, we assume
that every information frame has the same number of packets. Therefore, the total number of packets for the data part is:

\[ DATA = A \cdot n \]

**Simple Signature Scheme**

The total number of packets for the simple signatures in a cycle is:

\[ SIG_s = A \cdot \lfloor m/p \rfloor = A \cdot r \]

We use \( CYCLE_s \) to denote the length of a complete cycle for the simple signature scheme:

\[ CYCLE_s = SIG_s + DATA \]

The initial probe time is the period of time before the next signature arrives. Therefore, the average initial probe time is:

\[ PROBE_s = (r + n)/2 \]

After the initial probe period, the filtering process will last for a complete broadcast cycle. Therefore, the average access time is the sum of the initial probe time and the broadcast cycle.

\[ ACCESS_s = PROBE_s + CYCLE_s \]

\[ = (A + 0.5) \cdot (r + n) \]
Let $PT_s$ denote the period in which the mobile client is active during the initial probe time.

$$PT_s = \frac{r \cdot 1/2 \cdot r + n \cdot 1/2 \cdot n}{r + n} = \frac{r^2 + n^2}{2(r + n)}$$

To estimate the tune-in time, we have to first estimate the number of true matches. Let $P_s$ denote the selectivity of a query, the number of true matches is:

$$A_t = A \cdot P_s$$

In the filtering process, the mobile client has to tune in for all of the signature frames. In addition, it has to tune in for the true match and false drop information frames as well. Therefore, the tune-in time is:

$$TUNE_s = PT_s + SIG_s + A_t \cdot n + A_f \cdot n$$

$$= PT_s + SIG_s + A_t \cdot n + P_f^s \cdot A \cdot n - P_f^s \cdot A_t \cdot n$$

$$= PT_s + SIG_s + A \cdot n \cdot P_s + A \cdot n \cdot P_f^s - A \cdot n \cdot P_s \cdot P_f^s$$

$$= PT_s + SIG_s + DATA \cdot P_s + DATA \cdot P_f^s - DATA \cdot P_s \cdot P_f^s$$

**Integrated Signature Scheme**

Assume that $k$ information frames are grouped together in generating an integrated signature. The total number of integrated signatures is:

$$I = \lceil A/k \rceil$$
The total number of packets for the integrated signatures in a cycle is:

\[ SIG_i = I \cdot r = \left\lceil A/k \right\rceil \cdot r. \]

Therefore, the length of a complete broadcast cycle for the integrated signature scheme is:

\[ CYCLE_i = SIG_i + DATA \]

The average initial probe time is:

\[ PROBE_i = (r + k \cdot n)/2 \]

Similar to the simple scheme, the access time for the integrated scheme is:

\[ ACCESS_i = PROBE_i + CYCLE_i \]

Let \( PT_i \) denote the duration in which the mobile client is kept active during the initial probe time.

\[ PT_i = \frac{r \cdot 1/2 \cdot r + n \cdot k \cdot 1/2 \cdot n}{r + n \cdot k} = \frac{r^2 + n^2 \cdot k}{2(r + n \cdot k)} \]

The integrated scheme is good for broadcast with similarity among information frames, because similar information frames can be grouped together in generating the integrated signatures. Consequently, when a true match occurs, the frame group is likely to contain more than one qualified information frame. Let \( l \) denote the average number of qualified frames corresponding to a matched integrated signature.
The tune-in time for the integrated signature scheme is:

\[
TUNE_i = PT_i + SIG_i + I_t \cdot n \cdot k + I_f \cdot n \cdot k
\]

\[
= PT_i + SIG_i + \left[ \frac{A \cdot P_s}{l} \right] \cdot n \cdot k + P_f^i \cdot I \cdot n \cdot k
\]

\[
\approx PT_i + SIG_i + \frac{P_s \cdot k}{l} \cdot DATA + P_f^i \cdot DATA
\]

Note that \( P_f^i \) is a function of the number of bit strings superimposed, which we expect to be smaller than \( s \cdot k \).

**Multi-level Signature Scheme**

In this section, we assume a two-level signature scheme, which consists of integrated signatures at the higher level and simple signatures at the lower level. Thus, the total number of packets occupied by the signatures is:

\[
SIG_m = SIG_i + SIG_s
\]

The length of a complete cycle is:

\[
CYCLE_m = SIG_m + DATA
\]

The average initial probe time is derived based on the probability of the initial probe location and the corresponding probe time:

\[
PROBE_m = \frac{k \cdot n}{k \cdot n + k \cdot r + r} \cdot n + \frac{k \cdot r}{k \cdot n + k \cdot r + r} \cdot \left( \frac{r}{2} + n \right) + \frac{r}{k \cdot n + k \cdot r + r} \cdot \frac{r}{2}
\]

\[
= \frac{k \cdot n^2 + (k + 1)r^2 + 2 \cdot k \cdot nr}{2(k \cdot n + (k + 1)r)}
\]
Table 9: Parameter values of the cost models for simple, integrated and multi-level signature schemes.

<table>
<thead>
<tr>
<th>p</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>1000</td>
</tr>
<tr>
<td>s_s</td>
<td>100</td>
</tr>
<tr>
<td>s_i</td>
<td>280</td>
</tr>
<tr>
<td>k</td>
<td>4</td>
</tr>
<tr>
<td>l</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>10000</td>
</tr>
<tr>
<td>P_s</td>
<td>0.01</td>
</tr>
</tbody>
</table>

The access time can be approximated as follows.

\[ ACCESS_m = PROBE_m + CYCLE_m \]

The tune-in time in the initial probe period is:

\[
PT_m = \frac{k \cdot n}{k \cdot n + k \cdot r + r} \cdot \frac{n}{2} + \frac{k \cdot r}{k \cdot n + k \cdot r + r} \cdot \frac{r}{2} + \frac{r}{k \cdot n + k \cdot r + r} \cdot \frac{r}{2}
\]

To simplify the formula for the estimate of the tune-in time, we assume that the integrated signature for the group of the initial probe is a true match. Therefore, the tune-in time can be approximated as follows.

\[
TUNE_m = PT_m + SIG_i + k \cdot r + k \cdot P_s \cdot n + k \cdot P_j^s \cdot n + (I - \left[ P_s \cdot A/l \right] - 1) \cdot P_i^j \cdot (k \cdot r + k \cdot P_j^s \cdot n)
\]

Comparisons

Using the formulae developed above, we compare the access time and tune-in time of the three schemes. Table 9 lists the parameter values used in the comparisons. In the
comparisons, we assume that the selectivity of a query, $P_s$, is 1%. For the integrated and multi-level schemes, four information frames are grouped together to generate an integrated signature (i.e., $k = 4$). The false drop probabilities for the simple and integrated signatures, $P_j^s$ and $P_j^i$, can be calculated based on the length of a signature, $m$, and the numbers of bit strings superimposed for simple and integrated signatures, i.e., $s_s$ and $s_i$. Since the integrated scheme is good for broadcast with similar information frames, we assume that 30% of the superimposed bit strings are overlapped. Therefore, $s_i = (1 - 30\%) \cdot s_s \cdot 4 = 280$. Also, we assume that the locality of frames, $l$, is three. In other words, for the truly matched integrated signatures, three out of its associated information frames are truly qualified. The same assumption is also applied for the multi-level scheme. We vary the signatures length, $m$, from 1 to 35 packets to observe the changes on access time and tune-in time.

Figure 29 shows that the access times of the three schemes are linearly proportional to the size of the signatures. Since the overall size of the information frames is fixed at $10^7$ packets, the increase in access time over the signature size represents the overheads of the signature schemes. From the figure, we also find that the overhead of the multi-level scheme is close to the sum of the overheads for the other two schemes. This is attributed to the fact that the multi-level scheme is a combination of the other two schemes and that the overall size of signatures plays an important role to the increased access time.

Figure 30 shows the tune-in time for the three schemes. From the figure, we may observe that the tune-in time decreases to the minimal and then increases again as
Figure 29: Access time vs signature size.

Figure 30: Tune-in time vs signature size.
Table 10: Signature size, access time w.r.t. minimal tune time

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Minimal Tune-in Time</th>
<th>Access Time</th>
<th>Sig. Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>221620.82</td>
<td>10100505.00</td>
<td>10</td>
</tr>
<tr>
<td>Integrated</td>
<td>222370.82</td>
<td>10077015.00</td>
<td>31</td>
</tr>
<tr>
<td>Multi-level</td>
<td>142319.26</td>
<td>10138007.58</td>
<td>11</td>
</tr>
</tbody>
</table>

As the size of the signatures increase, from the data collected, we observe that at those minimal points the false drop probability is so low that false drops become insignificant to the tune-in time. Increasing the signature length further would only increase access time and tune-in time. The signature sizes and access time corresponding to the minimal tune-in time of the three schemes are listed in Table 10. The table shows the best signature sizes for the parameters used. From the table, we may estimate that the access delays for all three schemes are at most 1.4% greater than the broadcast cycle, while the tune-in time saved is more than 97.7%.

Although Table 10 gives us the best choice of the signature sizes for the three schemes, it doesn't give us a fair comparison for the performance among the three schemes. Figure 31 compares the tune-in times in terms of access time. With a given access time, the figure tells the best choice among the three schemes. From the figure, we may come to the conclusion that the integrated scheme and multi-level scheme are better than the simple scheme, while the integrated scheme is the best when access time is small and the multi-level scheme prevails otherwise. However, the performance of the integrated scheme is dependent on the locality in the frame groups and the overlaps among the superimposed bit strings. If the locality is 1 and
there is no overlap, the integrated scheme will have a worse tune-in time than the simple scheme due to longer initial probe time and heavy overhead for the true match frames (there is no overhead for true matches in simple scheme). In other words, the performance of the integrated scheme is highly dependent on the contents of the information frames and their clustering.

For the multi-level scheme, however, the signatures at the lowest level are indeed simple signatures. We expect the tune-in time of this scheme to be at least as good as the simple scheme with some access time delay for higher level signatures. Therefore, we compare the worst case of the integrated and multi-level schemes (i.e., the locality is 1 and there is no overlap of bit strings for information frames in a frame group) to

Figure 31: Tune-in time vs access time \((l = 3; s_t = 280)\).
the simple scheme. Figure 32 shows that with small access delay (i.e., less than 0.6% of the optimal access time) the tune-in time of the simple scheme and the multi-level scheme is roughly the same. For access delays of more than 0.6% of the optimal access time, the multi-level scheme has a much better tune-in time than the simple scheme.

![Graph showing tune-in time vs access time](image)

Figure 32: Tune-in time vs access time ($l = 1; s_i = 400$).

### 6.2.5 Adaptation of Signature Sizes

In the previous section, we fixed the size of the signatures used in the multi-level signature scheme. Since the integrated signatures index more information frames than the simple signatures do, the false drop probability of the integrated signatures is higher than that of the simple signatures. On one hand, since the integrated signa-
tures are more frequently used in filtering than the simple signatures, it is desirable to lower their false drop probability by increasing the signature length in order to reduce further matching of the associated simple signatures. On the other hand, lengthening the integrated signatures will increase the tune-in time and also the access time.

In this section, we fix the total overhead of the signatures and adjust the ratio of the storage occupied by the simple and integrated signatures in order to observe the minimal tune-in time for the multi-level signature scheme. To be consistent with our discussion in the previous section, we assume a two-level signature scheme. In the following, we use subscripts $i$ and $s$ to distinguish the symbols used for integrated signatures and simple signatures, respectively.

Assume that the size of the overall signature overhead is $SIG_m$. The portion of the overhead used for integrated signatures is $q$. Therefore,

$$SIG_i = q \cdot SIG_m$$

and

$$SIG_s = (1 - q) \cdot SIG_m$$

As a result, the numbers of packets allocated for each integrated signature and simple signature are: $r_i = \lfloor SIG_i / I \rfloor$ and $r_s = \lfloor SIG_s / A \rfloor$.

The cost formulae for the multi-level signature scheme are reformulated as follows. We only list the formulae which are different from that in Section 6.2.4.

$$PROBE_m = \frac{k \cdot n}{k \cdot n + k \cdot r_s + r_i} \cdot \frac{n}{2} + \frac{k \cdot r_s}{k \cdot n + k \cdot r_s + r_i} \cdot \frac{r_s}{2} + n + \frac{r_i}{k \cdot n + k \cdot r_s + r_i} \cdot \frac{r_i}{2} + n.$$
The tune-in time in the initial probe period is:

\[
PT_m = \frac{k \cdot (n + r_s)^2 + r_s^2}{2(k \cdot n + k \cdot r_s + r_i)}
\]

The tune-in time can be approximated as follows.

\[
TUNE_m = PT_m + SIG_i + k \cdot r_s + k \cdot P_s \cdot n + k \cdot P_f^s \cdot n + (I - \left[ P_s \cdot A/l \right] - 1) \cdot P_f^s \cdot (k \cdot r_s + k \cdot P_f^s \cdot n) + \left[ P_s \cdot A/l \right] (k \cdot r_s + l \cdot n + (k - l) \cdot P_f^s \cdot n)
\]

Comparison

Based on the cost models we developed above, we would like to see the change in tune-in time based on different ratios between the integrated and simple signatures. Figure 33 shows the tune-in time with respect to the share of the total signature overhead that the integrated signatures occupy, \( q \), given that the total signature overhead is 100,000 packets. We increase \( q \) from 0 to 1 by 1% each time. From the figure, the minimal tune-in time is at \( q = 0.3 \). In other words, when 30% of the total signature overhead are used for the integrated signatures, the tune-in time is the lowest. In this case, the sizes of the integrated and simple signatures are 12 and 7 packets per signature, respectively. However, the minimal point may change when a different total signature overhead is used. Since packets have fixed size, the allocated
signature space must be truncated to fit into an integral number of packets. This situation explains the zig-zag behavior in Figure 33.

Figure 34 compares, based on the same access time, the tune-in time of the "optimized" multi-level signature scheme and the original multi-level signature scheme (which has the same signature length for the integrated and simple signatures). In this experiment, we set the size of a frame group to 15. Other experiments with different frame group sizes have similar results. We observe that although the performance of the original configuration is not as good as the optimized one, the gap is quite insignificant. Therefore, one may choose to adopt the original configuration for the simplicity of signature generation and comparison.
6.3 Summary

Since most queries on broadcast information select only a small number of information frames, indexing is very effective in reducing the tune-in time. With a reasonable false drop probability and small signature overhead, signature schemes are excellent for information filtering in information broadcast services. Compared to traditional indexing, the signature methods are particularly suitable for mobile computers because they can perform realtime filtering with little processing and memory requirement.

This chapter discussed the application of signature techniques for information filtering. We proposed three signature schemes. We also analyzed the impact of the ratio between integrated signatures and simple signatures on optimizing tune-in time.
for the multilevel signature scheme.

The result of performance analysis shows that, with fixed signature size, the multilevel scheme has the best tune-in time performance but has the longest access time; the integrated scheme has the best average access time, but its tune-in time depends on the similarity among the information frames; the simple scheme has a fair access time and tune-in time. Compared to a broadcast channel without any indexing, all the three schemes improve tune-in time performance dramatically with a reasonable access time overhead.
CHAPTER VII

Signature Caching Techniques for Wireless Information Broadcast

7.1 Introduction

To reduce battery power consumption for mobile computers, we have to reduce the amount of information downloaded from the communication channel. Caching of frequently accessed information in mobile computers can reduce tune-in time considerably, and thus power consumption, because information can be fetched from the cache without tuning into the communication channel. Due to physical constraints, mobile computers usually have relatively small memory. Thus, caching large chunks of multi-media information frames is infeasible. Compared to the information frames, signatures need a rather small storage overhead and they contain critical information to support various kinds of queries. Thus, they are very suitable as the caching entity. In the following, we consider signature caching schemes for information broadcast and filtering services.

The two-level signature scheme is used in our discussion. There are two factors affecting signature maintenance in cache:
• Invalidation notice: An invalidation notice indicates which cached signatures are stale.

In order to support information filtering, the signatures in the cache have to be accurate. Therefore, the information server has to provide invalidation information to mobile computers. If some information frames and their corresponding signatures are updated, the information server will indicate the changes in the invalidation notice embedded in the broadcast. There are two kinds of invalidation notices, aiming at different situations.

– Bit tags: For each information frame, a 1-bit tag is allocated to indicate whether the corresponding information frame has been changed since the last broadcast cycle. The overhead of the bit tags is very low. However, the invalidation information only indicates changes with respect to the immediate preceding cycle. If a mobile computer has stopped tracking the invalidation information over a period of time, it cannot tell if the signatures in the cache is valid or not, even though the invalidation notice indicates no changes. Therefore, when a mobile client tunes into a channel, it has to reload the signatures into its cache memory in accordance with the caching policies used.

– Version numbers: To reduce the cost of reloading the signatures into the cache every time when a mobile computer tunes into a channel, multiple-bit version numbers may be used to serve as the invalidation notice. In the broadcast, a version number is assigned to each information frame. If
the information frame is modified, its version number is incremented by 1 (and reset to 0 when it reaches the largest number representable). The size of the version number is dependent on the frequency of updates on the information frames. However, it should not exceed that of the simple signatures, because, if it does, it will cost the mobile clients less by simply reloading all simple signatures. Due to the limited length allowed for the version numbers, the information server has to determine the period when the same version number won’t appear twice. The mobile clients will assume that the signatures they cached expire after they have lost track of the channel for the specific period of time and will reload the signatures in accordance with the caching policy used. If the mobile users listen to the broadcast channels before the version numbers have expired, they only need to reload the signatures which are changed during the off period.

- Refresh strategy: The refresh strategy determines which signatures to maintain in the cache.
  - Active refresh: Active refresh maintains all of the signatures in the cache. In order to maintain the accuracy of the signatures, a mobile computer has to load the updated signatures into the cache based on the invalidation notices. As a result, the performance of the active refresh strategy is influenced by the number of updates on the information frames.
  - Passive refresh: Passive refresh only keeps the previously accessed signatures in the cache. Instead of maintaining all of the signatures in the cache,
only the signatures received in the previous filtering process are kept in the cache. According to the invalidation notices, invalid signatures are cleared from the cache without refresh. Therefore, the performance of information filtering using passive refresh caching policies is not affected by the number of updates on the information frames.

7.2 Signature Caching Policies

In the following, we propose four signature caching policies based on the invalidation notice and cache refresh strategies. The cost models for tune-in time and access time are derived and their performance is compared. We assume that a two-level signature scheme is used in information broadcasting and filtering. The caching policies may be easily applied to other signature schemes. Table 11 lists the parameters and symbols used in our development of cost models for the signature caching policies.

7.2.1 Two-level Signature Scheme

In the following, we list the cost models for tune-in time and access time estimation of the two-level signature scheme [40].

The size of total signatures is:

\[ SIG = SIG_i + SIG_s, \]

where \( SIG_i \) and \( SIG_s \) are the number of packets for simple and integrated signatures, respectively.
Table 11: Parameters and symbols for cost models of signature caching policies.

\[ \begin{align*}
A & : \text{number of information frames in a broadcast cycle.} \\
A_f & : \text{number of information frames received due to false drops.} \\
A_t & : \text{number of information frames received due to true drops.} \\
I & : \text{number of integrated signatures in a broadcast cycle.} \\
I_f & : \text{number of integrated signatures matched as false drops.} \\
I_t & : \text{number of integrated signatures matched as true drops.} \\
P_f & : \text{false drop probability.} \\
P_f^s & : \text{false drop probability for simple signatures.} \\
P_f^i & : \text{false drop probability for integrated signatures.} \\
P_s & : \text{selectivity of a query.} \\
P_u & : \text{probability of updates on information frames.} \\
k & : \text{the number of information frames indexed by an integrated signature.} \\
l & : \text{locality of true drops (average number of true drops in a frame group).} \\
m & : \text{length of a signature in bits.} \\
n & : \text{the average number of packets in an information frame.} \\
p & : \text{the number of bits in a packet.} \\
r & : \text{the number of packets in a signature } (r = \lceil m/p \rceil). \\
s & : \text{the number of bit strings which are superimposed into a signature.} \\
t & : \text{the average number of cycles a mobile computer is disconnected from a channel.} \\
u & : \text{locality of updated information frames.} \\
v & : \text{size of version numbers in bits.}
\end{align*} \]

The length of a complete cycle is:

\[ CYCLE = SIG + DATA, \]

where \( DATA \) is the total size of the information frames in packets.

The average initial probe time is:

\[ PROBE = \frac{k \cdot n^2 + (k + 1)r^2 + 2 \cdot k \cdot nr}{2(k \cdot n + (k + 1)r)}. \]

The access time is:

\[ ACCESS = PROBE + CYCLE. \]
The tune-in time in the initial probe period is:

\[ PT = \frac{k \cdot n^2 + (k + 1)r^2}{2(k \cdot n + (k + 1)r)}. \]

The total tune-in time can be approximated as follows.

\[
TUNE = PT + SIG_i + k \cdot r + k \cdot P_s \cdot n + k \cdot P_f \cdot n \\
+ (I - [P_s \cdot A/l] - 1) \cdot P_f \cdot (k \cdot r + k \cdot P_f \cdot n) \\
+ [P_s \cdot A/l](k \cdot r + l \cdot n + (k - l) \cdot P_f \cdot n).
\]

### 7.2.2 Bit Tags and Active Refresh (BA)

The BA policy maintains all of the integrated signatures and simple signatures in the cache memory. For each frame group, we use bit tags in front of the integrated signatures to indicate the update status of the information frames in the group. When a bit tag signals that the corresponding information frame is changed, the integrated and simple signatures for the frame have to be loaded into the cache memory. The query signatures are then matched with the signatures maintained in the cache to decide which information frames may be skipped and which have to be brought into the mobile computers for false drop elimination. \( k \) bits are necessary for a group of \( k \) frames. The total broadcast overhead for invalidation information is:

\[ II_{BA} = \lceil k/p \rceil \cdot \lceil A/k \rceil. \]

Note that the invalidation information may share a packet with an integrated signature to make more compact use of the packets. Here we estimate its upper bound overhead.
The access time for the caching policies is:

\[ ACCESS = PROBE + II + CYCLE. \]

Since the formulae for access time is the same for the other policies, we will not repeat it in following sections.

For new users or those who lost track of the invalidation information, the tune-in time for loading signatures into the cache in the initial cycle is the number of frames for bit tags plus the number of signatures in a cycle:

\[ Init_{BA} = II_{BA} + SIG. \]

Since the cache loading process may be combined with the filtering process, a query may be answered while the signatures are being loaded into the cache. Therefore, the actual tune-in time is the sum of the initial probe time, the signature loading time, and the time to load the qualified information frames for false drop verification. Thus, the initial tune-in time is:

\[
Tune_{BA}^{init} = PT + Init_{BA} + (I - [P_s \cdot A/l]) \cdot P_f \cdot k \cdot P_f^i \cdot n + [P_s \cdot A/l](l \cdot n + (k - l) \cdot P_f^i \cdot n).
\]

For subsequent queries, the tune-in time is consumed for loading the modified signatures and listening to information frames for false drop elimination. The average number of modified integrated and simple signatures is \( P_u \cdot A \) and \([P_u \cdot A/u]\) respectively. The tune-in time for subsequent queries is:

\[
Tune_{BA}^{next} = PT + II_{BA} + P_u \cdot A \cdot r + [P_u \cdot A/u] \cdot r
\]
7.2.3 Version Numbers and Active Refresh (VA)

The VA scheme uses the version numbers of the information frames to inform mobile users of changes on the frames. Similar to bit tags, the version numbers for the information frames in a frame group are broadcasted before the integrated signatures. If the version number of a frame in the current broadcast is different from that of the same frame in the cache memory, we know that the signatures in the cache are not valid. Therefore, the updated signatures have to be brought into the cache. Due to the constraint on the size of the version numbers, the version numbers are valid only for a certain period of time. If a mobile computer loses track of the broadcast channel over that period of time, we will consider it as a new client for the channel. A new client has no prior knowledge of the channel and thus has to load the signatures into its cache in accordance with the caching policy. For an old client, however, only outdated signatures will be deleted and reloaded.

Assume that the size of a version number is $v$ bits. The period of time in which the version number is guaranteed valid is $2^v - 1$ broadcast cycles.

The total overhead for the invalidation information is:

$$II_{VA} = \lceil k \cdot v/p \rceil \cdot \lceil A/k \rceil.$$
For a new user to load signatures into the cache, the tune-in time is:

\[ \text{Init}_{VA}^{\text{new}} = II_{VA} + SIG. \]

For an old user who lost track of the broadcast channel for \( t \) cycles, where \( t < 2^v - 1 \), the average number of information frames which have not been changed during this period of time is \( A \cdot (1 - P_u)^t \). Therefore, the number of simple signatures which have to be reloaded is \( A - A \cdot (1 - P_u)^t \). Similarly, the number of integrated signatures which have to be reloaded is \( I - I \cdot (1 - \frac{[A \cdot P_u/u]}{I})^t \). Therefore, the signature loading time for an old user is:

\[
\text{Init}_{VA}^{\text{old}} = II_{VA} + (A - A \cdot (1 - P_u)^t) \cdot r + (I - I \cdot (1 - \frac{[A \cdot P_u/u]}{I})^t) \cdot r \\
= II_{VA} + SIG - SIG_s \cdot (1 - P_u)^t - SIG_i \cdot (1 - \frac{[A \cdot P_u/u]}{I})^t.
\]

If the initial cache loading process of a client is combined with the first query evaluation, the total tune-in time is:

\[
\text{Tune}_{VA}^{\text{init}} = PT + \text{Init}_{VA}^{\text{new}} + (I - [P_s \cdot A/l]) \cdot P_j \cdot k \cdot P_j \cdot n \\
+ [P_s \cdot A/l] (l \cdot n + (k - l) \cdot P_j \cdot n),
\]

where \( \text{Init}_{VA}^{\text{init}} \) is either \( \text{Init}_{VA}^{\text{new}} \) or \( \text{Init}_{VA}^{\text{old}} \) depending on whether the user is new to the channel or not.

For subsequent queries, the tune-in time is consumed for loading the modified signatures and listening to information frames for false drop elimination. The tune-in
time for subsequent queries is:

\[ T_{\text{next}}^{VA} = PT + II_{VA} + P_u \cdot A \cdot r + [P_u \cdot A/u] \cdot r \]
\[ + (I - [P_s \cdot A/l]) \cdot P_f \cdot k \cdot P_n \cdot n \]
\[ + [P_s \cdot A/l](l \cdot n + (k - l) \cdot P_f \cdot n). \]

### 7.2.4 Bit Tag and Passive Refresh (BP)

In the BP policy, instead of caching all of the signatures in the mobile computer, only the signatures received in previous queries are kept in the memory. Instead of actively updating all of the signatures in the cache memory to maintain their accuracy, the outdated signatures are deleted from the cache in accordance with the bit tags. New signatures are brought into the cache only if they are needed in the current filtering process. Like the BA policy, the bit tags corresponding to a frame group are broadcasted before the integrated signature. If the bit tags show no change in the frame group, the integrated signature in cache will be used for processing the query. If the bit tags indicate changes in some information frames, the corresponding integrated and simple signatures in cache are deleted and the updated integrated signature is loaded into the cache for comparison with the integrated query signature. If the comparison between the query signature and the integrated signature is a match, the simple signatures not residing in the cache will be loaded into the cache and compared to the simple query signature. If the comparison with the integrated signature fails,
the mobile computer simply tunes off without refilling the cache for the deleted simple signatures.

The total number of frames needed for the BP scheme is:

\[ II_{BP} = \left\lceil k/p \right\rceil \cdot \left\lceil A/k \right\rceil. \]

Because of the passive manner of signature caching adopted by the BP policy, we don’t consider the tune-in time needed for loading signatures. Since we don’t bring all of the signatures into the cache, the tune-in time for the initial query is the same as answering a query without caching plus the overhead for the bit tags.

\[
Tune_{\text{BP}}^{\text{init}} = PT + SIG_t + (I - \left\lfloor P_s \cdot A/l \right\rfloor) \cdot (P_j^i \cdot k \cdot r + P_j^s \cdot k \cdot P_j^s \cdot n) + \left\lfloor P_s \cdot A/l \right\rfloor (k \cdot r + l \cdot n + (k - l) \cdot P_j^s \cdot n).
\]

In the initial cache loading cycle, the average number of signatures received and maintained by the mobile computer is

\[ I + (I - \left\lfloor P_s \cdot A/l \right\rfloor) \cdot P_j^i \cdot k + \left\lceil P_s \cdot A/l \right\rceil \cdot k. \]

Let \( I_{\text{cache}} \) and \( A_{\text{cache}} \) be the number of integrated and simple signatures in cache, respectively. Then,

\[ A_{\text{cache}} = (I - \left\lfloor P_s \cdot A/l \right\rfloor) \cdot P_j^i \cdot k + \left\lceil P_s \cdot A/l \right\rceil \cdot k. \]

In this scheme, every integrated signature has been maintained in cache because every integrated signature has to be checked in order to decide whether the corresponding simple signatures and frame groups may be skipped. Thus,

\[ I_{\text{cache}} = I. \]
The number of integrated signatures deleted from and brought into the cache is the same as the number of integrated signatures modified: \([P_u \cdot A/u]\). The average number of simple signatures purged from the cache is the number of simple signatures in cache times the percentage of updates, i.e., \(P_u \cdot A_{cache}\). The simple signatures brought into the cache are those corresponding to successful matches between the integrated signatures and integrated query signature but not residing in the cache. Thus, the number of simple signatures brought into the cache are: \(((I - [P_s \cdot A/l]) \cdot P_f^i \cdot k + [P_s \cdot A/l] \cdot k) \cdot (1 - \frac{A_{cache}}{A} \cdot (1 - P_u))\).

For subsequent queries, the average tune-in time is:

\[
Tune_{BP}^{next} = PT + II_{BP} + [P_u \cdot A/u] \cdot r + ((I - [P_s \cdot A/l]) \cdot P_f^i \cdot k + [P_s \cdot A/l] \cdot k) \\
\cdot (1 - \frac{A_{cache}}{A} \cdot (1 - P_u)) \cdot r + (I - [P_s \cdot A/l]) \cdot P_f^s \cdot k \cdot P_f^s \cdot n \\
+ [P_s \cdot A/l] \cdot (l \cdot n + (k - l) \cdot P_f^s \cdot n).
\]

### 7.2.5 Version Numbers and Passive Refresh (VP)

The VP scheme uses the version numbers of the information frames to inform mobile computers of changes in the frames and to refresh the cache in a passive manner. Assume that the size of a version number is \(v\) bits. The total broadcast overhead for the invalidation information is:

\[
II_{VP} = [k \cdot v/p] \cdot [A/k].
\]

Similar to the BP policy, new signatures are brought into the cache only when they are needed in the current filtering process. The tune-in time for the initial query
is the same as for answering a query without caching plus the overhead for version numbers. However, depending on whether the mobile computer is new to the channel, the tune-in time for the initial query will be different. For a new client, the initial tune-in time is:

\[
T_{\text{init,new}} = PT + II_{VP} + SIG_i + (I - [P_s \cdot A/l]) \cdot (P^i \cdot k \cdot r + P^i \cdot k \cdot P^s \cdot n) + [P_s \cdot A/l](k \cdot r + l \cdot n + (k - l) \cdot P^s \cdot n).
\]

Assume that the number of integrated and simple signatures cached in a mobile client before it tunes off is \(I_{\text{cache}}\) and \(A_{\text{cache}}\), respectively. In the two-level signature scheme, since every integrated signature is examined, a complete set of the integrated signatures has to be kept in the cache. Therefore, \(I_{\text{cache}} = I\). For an old client which has been tuned off for \(t\) cycles, the number of valid simple signatures in cache is \(A_{\text{cache}} \cdot (1 - P_u)^t\) and the number of valid integrated signatures in cache is \(I \cdot (1 - \frac{A \cdot P_u}{I})^t\). As a result, the tune-in time for an old user's initial query is:

\[
T_{\text{init,old}} = PT + II_{VP} + (I - I(1 - \frac{A \cdot P_u}{I})) \cdot r + (I - [P_s \cdot A/l]) \cdot P^i \cdot k \cdot r(1 - \frac{A_{\text{cache}}}{A}(1 - P_u)^t) + (I - [P_s \cdot A/l]) \cdot P^i \cdot k \cdot P^s \cdot n + [P_s \cdot A/l] \cdot k \cdot r(1 - \frac{A_{\text{cache}}}{A}(1 - P_u)^t) + [P_s \cdot A/l](l \cdot n + (k - l) \cdot P^s \cdot n).
\]
Table 12: Parameters values for the cost models of signature caching policies.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td>4</td>
</tr>
<tr>
<td>$l$</td>
<td>1</td>
</tr>
<tr>
<td>$m$</td>
<td>1280</td>
</tr>
<tr>
<td>$n$</td>
<td>1000</td>
</tr>
<tr>
<td>$p$</td>
<td>128</td>
</tr>
<tr>
<td>$r$</td>
<td>10</td>
</tr>
<tr>
<td>$s$</td>
<td>100</td>
</tr>
<tr>
<td>$u$</td>
<td>1</td>
</tr>
<tr>
<td>$v$</td>
<td>10</td>
</tr>
<tr>
<td>$A$</td>
<td>10000</td>
</tr>
<tr>
<td>$P_s$</td>
<td>0.01</td>
</tr>
<tr>
<td>$P_u$</td>
<td>0.01</td>
</tr>
</tbody>
</table>

After the initial cycle, there is no difference between new and old clients. Therefore, the average tune-in time for subsequent queries is:

$$T_{\text{next}}^{\text{Tune}} = PT + II_{VP} + \left[P_u \cdot A/u\right] \cdot r$$

$$+ (I - \left[P_s \cdot A/l\right]) \cdot P_f^i \left(1 - \frac{A_{\text{cache}}}{A} \cdot (1 - P_u)\right) \cdot k \cdot r$$

$$+ \left[P_s \cdot A/l\right] \left(1 - \frac{A_{\text{cache}}}{A} \cdot (1 - P_u)\right) \cdot k \cdot r$$

$$+ (I - \left[P_s \cdot A/l\right]) \cdot P_f^i \cdot k \cdot P_f^s \cdot n$$

$$+ \left[P_s \cdot A/l\right] (l \cdot n + (k - l) \cdot P_f^s \cdot n).$$

7.3 Comparison of Signature Caching Policies

There are several factors, such as the percentage of updates, size of signatures, and number of information frames in a group, affecting the tune-in time of a caching policy. In the following, we vary these factors to observe their influence on the caching policies.

First, we compare the tune-in time of the caching policies for the first ten queries after a mobile user connects to a broadcast channel. Important parameter values used
in the comparison are listed in Table 12. We fix the signature size, \( r \), to 10 packets and the average percentage of frames updated in each cycle, \( P_u \), to 1%. The locality of modified frames in a group, \( u \), is 1. Based on the formulae developed in [40], the false drop probabilities for integrated signatures and simple signatures, i.e., \( P_f^I \) and \( P_f^S \), can be derived.

![Figure 35: Tune-in time vs query number.](image)

We use the tune-in time of a cacheless two-level scheme as reference. In Figure 35, the symbols for different policies are self-explanatory except that we add N and O in front of VA and VP to distinguish the cases for new and old users. New users have no signatures in cache while old users have some signatures left in the cache before they were switched off. For the old users, we assume that they have disconnected for
four broadcast cycles. Since the bit tag methods use only one bit per frame, it would be unfair to assume that the bit tags occupy an entire packet. Therefore, we combine the bit tags with the integrated signatures in the evaluation. On the other hand, the length of a version number is set to 10 bits.

Figure 35 shows the tune-in time of the policies for the initial and subsequent queries. In general, the tune-in time of the initial query is much higher than that of subsequent queries, because signatures have to be loaded into the caches in the initial query. Furthermore, only the BA and NVA policies have dramatically higher tune-in time than the cacheless two-level scheme, because they have to load all of the integrated and simple signatures into the cache. OVA and OVP have the best tune-in time, because they have signatures left in the cache from the previous connection. OVA is better than OVP, because OVA has more signatures left in the cache. On the other hand, the tune-in time of the passive policies for new users is close to the cacheless scheme, because the passive policies are dependent on the signature filtering process.

For subsequent queries, the BA policy has the best performance, while NVA and OVA, which have identical tune-in time, are close behind it. The difference between the BA and VA policies is due to the overhead of the invalidation information. In this comparison, the active policies outperform the passive policies. However, for subsequent queries, the passive policies gradually catch up. Also note that, the update rate is set to 1%, which is rather low. As we will show later, the update rate plays an important role in determining whether active and passive policies are best.
From the data, all the caching policies have better average tune-in time than the cacheless method after only listening to the channel for three broadcast cycles. Thus, we conclude that the signature schemes in general will benefit from the caching techniques.

Next, we compare the tune-in time of the caching policies by changing the size of the signatures. For this comparison, we calculate the average tune-in time for the initial query and for two subsequent queries. To simplify our comparison, we use the average tune-in time for old users in the version number policies.

Figure 36 shows that VA < VP < BP < BA. In the figure, the tune-in time of BA is higher than that of the cacheless approach as the signature size increases. In
practice, however, the signature size is chosen to yield the best performance of a caching policy. Therefore, the comparison should emphasize the minimal points of the policies.

![Graph showing tune-in time vs access time](image)

Figure 37: Tune-in time vs access time.

The above comparison shows the best performance of the policies. However, it didn't show their overhead on access time. In Figure 37, we show the average tune-in time of the first three queries against the access time for the proposed caching policies. For the cacheless, index-free filtering method, tune-in time and access time are equal to the broadcast cycle, i.e., 10,000,000 packets with the parameters we set. As shown in the figure, the tune-in time for all of the cacheless and cached signature filtering methods rapidly decreases until the access time delay is at around 100,000
to 150,000 packets. We also observe that the cached signature filtering methods have better tune-in time than the cacheless method when more than 75,000 packets of access time delay is allowed. Also note that, the tune-in time performance of the cached signature filtering methods is comparable to the cacheless signature scheme when the access time delay is less than 75,000 packets. As the number of subsequent queries increases, the tune-in time for the caching policies will be lower. If enough subsequent queries are included, the caching policies will be absolutely better than the cacheless method (i.e., no cross-over point).

In addition to the factors discussed above, the number of information frames grouped together to generate an integrated signature also affects tune-in time performance in two ways. On one hand, increasing the number of frames grouped together allows more compact storage of the invalidation information in packets and thus reduces its overhead; on the other hand, the information abstracted and stored into the integrated signatures will increase, thus resulting in higher false drop rates. Figure 38 compares the caching methods based on the group size. In order to more easily observe the impact of the group size on fitting invalidation information into packets, we don’t combine the bit tags with integrated signatures as we did in the previous comparisons. To reduce the overhead of storing a bit tag in a packet, we use a packet size of 4 bytes. Thus, the signature size is changed to 32 packets in order to keep the same false drop probability with the previous comparisons. In the figure, as the group size increases, we find that the tune-in time of the bit tag policies drop initially but then increase and remain at the same levels. The initial drop is due to the more
compact storage of the bit tags and the reduced number of integrated signatures. However, the factor of false drops dominates later on. The version number policies perform better mainly because they have inherited cached signatures from previous connections.

![Tune-in time vs group size](image)

Figure 38: Tune-in time vs group size

Finally, we vary the percentage of updated frames to observe its effect on caching policies. In this comparison, we set the locality of the modified frames in a frame group to 2 in order to allow the percentage of the updated frames to go up to 50%. Figure 39 shows the tune-in time performance. As the update percentage increases, the caching methods using passive refreshing strategy perform better than the methods using active strategy. This is because the passive strategies only load the signatures
needed for query processing but not residing in the cache, while the active policies try to maintain all of the signatures in cache.

7.4 Limiting The Cache Size

In the previous sections, we assumed that the cache size is large enough for accommodating all of the cached signatures. However, some mobile computers may have rather small memory for caching signatures. Due to physical constraints, replacement policies have to be considered for mobile computers with small cache memory.

7.4.1 Cache Replacement Policies

Two factors of the broadcast and filter services have to be considered when we design replacement policies: access frequency and update frequency. Update frequency
is determined by the information servers at the fixed network. However, the mobile computers may estimate it, along with the access frequency, using a statistical approach.

Due to the sequential scanning nature of the signature schemes, all the integrated signatures have to be compared with the integrated query signature in order to decide whether their associated simple signatures are needed for subsequent comparisons. Therefore, user access patterns do not have an impact on the cost of integrated signature comparison. As such, a simple but feasible cache replacement policy is to keep as many integrated signatures in cache as possible. When a signature is updated, its slot is filled with the next integrated signature not in cache. The policy is to let some infrequently updated integrated signatures eventually get into the cache.

Intuitively, the above policy makes full use of the cache available when the cache size is not large enough to hold all of the integrated signatures. When the cache size is large enough, the simple signatures should be considered for caching too, especially those corresponding to frequently matched integrated signatures.

To cache a mix of integrated and simple signatures, we adopt the policy which replaces the most frequently updated and least frequently accessed signatures. Compared to simple signatures, integrated signatures are more frequently accessed than most of the simple signatures. However, integrated signatures also have a higher average update rate than simple signatures do. The reason is that when any information frame in the frame group is updated, the corresponding integrated signature has to be updated while only the simple signature corresponding to the updated information
frame in the group has to be updated. Therefore, an integrated signature may be moved out of the cache while some of the simple signatures in its frame group may stay in the cache.

The approach we proposed for general signature caching combines a passive refresh policy and the replacement policy described above. The refresh strategy is to decide when to execute the replacement policy, while the replacement policy is to decide which information frames should stay in the cache. Based on the passive refresh policy, a signature will be considered for caching only when it is received over the channel. For the replacement policy, we maintain a replacement score (RS) for each signature to decide which signatures are to be cached.

Table 13: Actions for cache replacement.

<table>
<thead>
<tr>
<th>Updated?</th>
<th>In Cache?</th>
<th>To compare?</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>go to doze mode.</td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td>update replacement score and go to doze mode.</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td></td>
<td>go to doze mode.</td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td>delete the signature from cache; update replacement score and go to doze mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+</td>
<td>receive the signature over the channel, do comparison, update replacement score and execute the cache replacement policy.</td>
</tr>
<tr>
<td>+</td>
<td></td>
<td>+</td>
<td>do comparison on cached signature; update replacement score and go to doze mode.</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
<td>receive the signature from the channel, update the cache, do comparison and update replacement score.</td>
</tr>
</tbody>
</table>
When a mobile computer starts the filtering process, it will synchronize to the broadcast channel. From then on, for each incoming signature, it checks whether the signature is updated, in cache, and needed for signature comparison. Table 13 describes the actions taken under various situations.

The cache replacement algorithm maintains a replacement score (RS) for each signature in the broadcast cycle:

\[ RS = \text{number of comparisons} - \omega \cdot \text{number of updates}. \]

where \( \omega \) is the weight of update to comparisons and \( 0 \leq \omega \leq \infty \).

The replacement scores are updated when a signature is updated or is received over the channel for comparison. When a signature is received over the channel, it will be considered for caching. Its replacement score is compared with the lowest score of the cached signatures. If the score is higher than the lowest score, the signature is brought into the cache to replace the signature with the lowest score.

### 7.4.2 Performance Evaluation of the Cache Replacement Policy

In the following, we simulate the signature cache replacement policies. In the simulation for the policy of caching both simple and integrated signatures, we assume the weight for the number of updates, \( \omega \), to be 1. In our experiments, we fix the size of a frame group, \( k \), to 4. We assume the selectivity of queries, \( P_s \), to be 5%, update probability, \( P_u \), to be 1%, and false drop probability of the integrated signatures, \( P_f^i \), to be 1%. The localities for true drops and updates, i.e., \( l \) and \( u \), are set to 2. Initially, the cache is empty, but it is gradually filled in subsequent cycles and they
have increasing hit rates. In the simulation, we measure the number of hits from the initial query up to the 25th query. To observe the effect of cache size on the number of hits, we repeat the same experiment for various cache sizes.

![Figure 40: Hits vs query number.](image)

Figure 40 shows the number of hits for different cache sizes. On the top of the figure, the curve labeled as 'reference' is the number of signatures needed for comparisons. In other words, it represents the number of signatures received from the communication channel if no cache is used. Other curves are labeled by the policy and cache size used. For example, 'INT 1000' represents the cache hits for the pure integrated signature cache replacement policy with cache size of 1000 signatures, whereas 'MIX 2000' represents the cache hits for the mixed signature cache replacement policy with cache size of 2000 signatures. It can be observed that in every configuration, the
initial query has no cache hit, but the second query rapidly increases the hits. Then, the cache hits for the MIX experiments gradually increase for subsequent queries, while the cache hits for INT experiments remain at the same levels. In the figure, 'INT 3000' and 'INT 4000' are overlapped, because there are only 2500 integrated signatures in our experiment. Therefore, the INT method with cache size greater than 2500 signatures has the same cache hit results.

![Figure 41: Hits vs cache size.](image)

Figure 41 shows the average number of hits over the 25 queries for different cache sizes. From the figure, we observe that, for the INT method which caches only the integrated signatures, the number of cache hits is proportional to the cache size up to 2500, the total number of integrated signatures used in the simulation. From then on, increasing the cache size makes no difference, because all of the integrated
Signatures are already cached. On the other hand, for the MIX method, the number of hits increases steadily and is proportional to the cache size. When the cache size is over 2500, however, the increase slows down but continues. The phenomenon may be explained as follows. At the beginning, as many integrated signatures as possible will be brought into the cache. Since integrated signatures are always used for comparison, the hit rate is high. As most of the integrated signatures are in cache, the number of simple signatures in cache increases. Whether a simple signature is used in the filtering process depends on the selectivity and the false drop probability of the integrated signatures. Therefore, the hit rate of the simple signatures is not as high as that of the integrated signatures. Thus, the increase of hits slows down when the cache size is greater than 2500. The same reason also explains why the MIX method has slightly lower hit rates than the INT method when the cache size is smaller than 2500 signatures.

7.5 Summary

This chapter discusses the policies for caching signatures with mobile computers. Based on the invalidation information and cache refresh strategies, we studied four signature caching policies with analytical cost models.

The performance evaluation shows that, with reasonable access time delay, the two-level signature scheme with various caching policies outperforms the same scheme without caching. The policies using version numbers are in general better than those using bit tags. As the percentage of updated frames increases, using a passive re-
freshing strategy is better than using an active strategy.

Moreover, we discuss the cache replacement policies for mobile computers with small caches. Two replacement policies are proposed, one caches integrated signatures only and the other caches both of the simple and integrated signatures. Cost models for the signature schemes and caching policies are derived in terms of access time and tuning time. For small cache, the replacement policy caching only integrated signatures is better, while the policy caching both kinds of the signatures has comparable performance. On the other hand, when the cache size is larger than the size of the integrated signatures, the policy caching both kinds of signatures is better.
CHAPTER VIII

Summary and Future Directions

In this thesis, we describe indexing techniques for object-oriented database systems and mobile computing systems. This chapter summarizes the main research results and concludes with a discussion of future research directions.

8.1 Summary

Query processing is one of the most important implementation issues for database systems. Indexing techniques have played an important part in the success of query processing for traditional database systems such as relational database systems. We can envision that indexing techniques will also play a critical role in the success of nontraditional database systems such as object-oriented database systems and mobile computing systems. We studied indexing techniques for query processing in object-oriented database systems and for information broadcast and filtering in mobile computing systems.

8.1.1 Object-Oriented Database Systems

For the object-oriented database systems, we focused on the processing of nested object queries. We first described a basic object data model to introduce aggregation
hierarchy and nested object queries. We then provided a classification of nested object queries based on 1) the relative positions of the target and predicate classes on the aggregation hierarchy; and 2) the complexity of the predicates and operators used in the predicates. Examples for each type of the nested query were also given. After examining traditional traversal approaches for processing nested queries and surveying several proposed indexing techniques in the literature, we identified some areas for improvement.

To improve the processing of nested object queries, three new organizations based on signature file methods and index methods were presented in this thesis. For each method we proposed, we discussed the design and implementation of the organizations and provided the analytical cost models for performance evaluation. Storage overhead and cost for retrievals and updates were used as the criteria for performance evaluation.

Signature file methods use abstracted information in the signature file to avoid actual retrieval of intermediate objects located on the path from top classes to the nested attributes. The purpose of using a signature file is to screen out most of the unqualified objects. An object signature failing to match the query signature guarantees that the corresponding object can be ignored. Therefore, unnecessary object accesses are prevented. Two signature file schemes for nested object query processing were presented, namely, tree signature and path signature. We described the algorithms for query processing and signature file maintenance involving update operations. Cost models were also provided to facilitate the analysis of overhead cost
and performance of the signature schemes. Signature file schemes have a much simpler file structure than inverted indexes. They are particularly good for multi-attribute retrieval when the attributes have an equal chance of being specified in the query.

Moreover, we introduced two new index methods, the path dictionary index and the direct link index, to support efficient nested query evaluation in object-oriented databases. An object-oriented database may be viewed as a space of objects connected with links through complex attributes. These two methods extract the complex attributes from the database to represent the connections between objects. Since primitive attribute values are not stored in the path dictionary indexes, it is much faster to traverse the nodes in the indexes than the objects in the database. Therefore, the path dictionary index and direct link index can be used to reduce the number of accesses to the database, and, in particular, to avoid accessing intermediate objects when traversal from one class to another is performed.

The common property of these two organizations is that they provide secondary access structures for the object databases. Thus, the addition of the index organizations will not affect the normal query processing of the databases. These two organizations also support associative search on an arbitrary number of attributes with a small amount storage expense. The path dictionary index supports efficient evaluation of queries involving any object on a given path of classes, while the direct links index supports efficient evaluation of queries involving any object in two arbitrary classes. The concepts and implementations of the path dictionary index and direct links index were presented in this thesis. For these two organizations, cost
models were derived and performance evaluations were conducted.

We compared the storage overhead and the retrieval and update costs of the path dictionary index to those of the well-known path index. Our results show that the path dictionary index is superior to the path index both in terms of storage overhead and in terms of retrieval and update costs.

In contrast to a path dictionary index, a direct link index is effective in providing short cuts for object traversal. Based on direct links, a new class of indexing mechanisms, forward index and backward index, for both forward and backward traversals were proposed. Because of the clustering of direct links, a forward or backward index can be realized in one of the following ways: clustering index, semi-clustering index, or non-clustering index. Our analysis shows that the clustering index implementation has the best performance in storage overhead and retrieval speed. A Semi-clustering index has the same storage overhead as a non-clustering index, but its retrieval speed is faster. Although a non-clustering index is not as good as the other two implementations, it is still much better than the traditional traversal approaches.

8.1.2 Mobile Computing

For mobile computing, indexing methods for information broadcast and filtering services on wireless channels were presented. The signature techniques are natural approaches for indexing the information broadcast and filtering services, because the signature files are easy to be "linearized" and "distributed" for broadcasting on the air. We presented three signature schemes, namely, simple signature, integrated sig-
nature, and multi-level signature, in this thesis. Methods on how to disseminate the signatures and information on the air were described.

Access time and tune-in time were used as the criteria for evaluating the performance of signature schemes for information broadcast and filtering. Access time describes how much time the users have to wait to obtain the interesting data items. Because it correlates with battery consumption, the second measure is tune-in time, i.e., the period of time when the mobile computers have to actively monitor the wireless channels. With a reasonable false drop probability and small signature overhead (in terms of access time), the signature schemes are excellent in reducing the tune-in time.

The cost models for tune-in time and access time of each of the signature schemes was developed and compared based on various factors. The result of performance evaluation shows that, with fixed signature size, the multilevel scheme has the best tune-in time performance but has the longest access time; the integrated scheme has the best average access time, but its tune-in time depends on the similarity among the information frames; the simple scheme has a fair access time and tune-in time. Compared to a broadcast channel without any indexing, all of the three schemes improve tune-in time performance dramatically with a reasonable access time overhead.

In addition to utilizing the signature techniques for information broadcast and filtering services, we also proposed methods for caching signatures in mobile hosts. Taking invalidation and refresh strategies of signatures in cache into consideration,
four signature caching policies for the multi-level signature scheme were presented. The signature caching policies further reduce tune-in time of the mobile hosts for the information broadcast and filtering services. Like our analysis on signature schemes, cost models for signature caching policies were provided to facilitate the performance evaluation of the signature caching policies. Our analysis and comparisons show that, with reasonable access time delay, the multi-level signature scheme with various caching policies outperforms the same scheme without caching. Policies using version numbers for invalidation of cached signatures are in general better than those using bit tags. As the percentage of updated frames increased, policies using passive refreshing strategy are better than those using active strategy.

In mobile environments, some of the computers may have rather small memory for caching signatures. Due to physical constraints, replacement policies have to be considered for mobile computers with small cache memory. To address this issue, we studied two cache replacement policies for the multi-level signature scheme with version number invalidation and passive refresh strategy. One of the replacement policies caches the integrated signatures only and the other policy caches both the integrated and the simple signatures. The performance evaluation of the replacement policies were conducted by simulation. We compared these two policies in terms of cache hits. The result shows that, for small cache, the policy caching only integrated signatures is better, while the policy caching both kinds of the signatures has a comparable performance. On the other hand, when the cache size is larger than the total size of integrated signatures, the policy caching both kinds of signatures is
better.

8.2 Contributions

We studied indexing techniques for query processing of nontraditional database systems, namely, object-oriented database systems and information broadcast and filtering services in mobile environments. The development of these indexing techniques helps us to understand and solve the problems of nested object query processing in object-oriented database systems and information broadcast and filtering services in mobile environments. The proposed methods have been evaluated thoroughly based on important performance measures.

The main contributions of this thesis are the following:

1. A classification of nested object queries (Chapter 2) which
   - facilitates the understanding and discussion of the characteristics of nested object queries,

2. Two signature file schemes, tree signature and path signature, for query processing in OODBs (Chapter 3). These schemes
   - are the first signature methods proposed for indexing object-oriented database systems,
   - have simple file structures with small storage overhead,
   - support queries involving arbitrary or specific attributes in the database.
3. Two organizations, path dictionary and direct links, for indexing objects in an aggregation hierarchy of classes (Chapter 4). The methods

- demonstrate the impact of path information in query processing,
- provide an encoding scheme, s-expression, for path information,
- support a variety of nested queries efficiently,
- support extensibility to indexing organizations,
- provide significant retrieval performance improvement with small storage and update overhead.

4. Three signature file schemes, simple signature, integrated signature and multi-level signature, for mobile computing systems (Chapter 6). These methods

- are the first signature methods proposed for indexing the information broadcast,
- demonstrate that signature schemes are natural approaches for information broadcast and filtering services,
- reduce tune-in time (i.e., battery power) dramatically with small access time delay,
- address the issue of the optimization of signature sizes for multi-level signature scheme.

5. Caching schemes for mobile computing systems using the multi-level signature scheme (Chapter 6) which
• characterize the issue of caching signatures for information broadcast and filtering services,
• provide policies to address the issue of signature caching,
• further reduce tune-in time for information broadcast and filtering services,
• address the issue of signature cache replacement for mobile hosts with small cache.

8.3 Future Directions

There are a number of research issues resulting from this thesis.

We have shown that tree-structured indexing techniques, e.g., B+-Tree, can be combined with path dictionary to improve significantly the performance of nested object queries in object-oriented databases. A new method which combines the signature file technique with the path dictionary presents a future research topic on nested query processing.

Our study on path dictionary was focused on aggregation hierarchy without considering recursive relationships and methods. It will be interesting to explore extension of the path dictionary organization to cover recursive path information and object connections realized by methods. Moreover, we assumed that the path dictionary for a database with a hierarchical schema may be decomposed into smaller path schemas in order to simplify the design of the path dictionary. Thus, plans for decomposing the aggregation hierarchy into paths in order to reach optimal performance for the
overall system deserve a thorough study. New strategies for query optimization using the cost models we developed for the path dictionary may be developed.

A distributed computing environment imposes new challenges for processing nested object queries. There are many issues on nested query processing in a distributed OODBS that need to be explored, e.g., system configuration, distribution of objects, format of object identifiers, query processing strategies and indexing techniques. In [39], we proposed query processing strategies and supporting mechanisms for nested queries in distributed object-oriented database systems. We argued that the indexing mechanisms are very important in a distributed environment due to data distribution and communication costs. We discussed several approaches to using indexing organizations and path dictionary in nested query processing and felt that the path dictionary is the most suitable secondary organization for the distributed environment. Thus, it will be an interesting project to extend the path dictionary for a distributed framework such as COBRA [44]. Moreover, the issues of partitioning and replicating the path dictionary for distributed environments need to be addressed. The in-memory management of the path information has a great impact on the performance of distributed object management systems. Different indexing organizations and methodologies for determining the configuration of the index organizations need to be investigated.

Information broadcast and filtering services demonstrate the usage of public (broadcast) channels. An on-going project on the channel management problem is to combine broadcast channels and on-demand (point-to-point) channels in order to optimize
the overall system performance. Moreover, the global channel management problem for neighboring cells, in which channel borrowing is allowed, is a great challenge.

Use of the public broadcast channels as shared cache for mobile clients is an interesting idea. A coherent approach for the caching, invalidation and maintenance of data items through a combination of public and on-demand channels also needs further research.

Finally, a framework of mobile computing systems, which allows dynamic management of various types of channels, provides information brokerage facilities, and properly handles information consistency and security issues, represents a great challenge. With a foundation of this framework, an architecture of a service agency which efficiently utilizes both broadcast channels and on-demand channels may be developed. Various system issues for the mobile computing services, such as caching, disconnections, and location management, then can be investigated.
APPENDIX A

Proof of Lemmas

Lemma 6.2.4.1 Given the length of a signature in bits, $m$, the number of bit strings superimposed into the signature, $s$, and the number of 1's set in the bit strings, $w_b$, the average number of 0's in the signature is:

$$\bar{w}_f \approx me^{-w_b s/m}.$$  

Proof of Lemma 6.2.4.1: Assume that a good hash function is used so that each of the potential bit strings has the same probability of being used in generating frame signatures. Given that a bit string has $m$ bits of which $w_b$ bits are set to 1, The total number of combinations for the signature generated by superimposing $x$ bit strings is \( \binom{m}{w_b}^x \).

We consider a particular set of $y$ bits in a frame signature. Let $p(x, y, i)$ be the probability for the frame signature in which $i$ of the $y$ bits are set to 0. Since a bit in a frame signature is 0 if and only if the corresponding bit in each of the $x$ bit strings are 0. Therefore,

$$p(x, y, i) = \frac{\left( \frac{m - i}{w_b} \right)^x}{\left( \frac{m}{w_b} \right)^x}.$$  

188
Thus, the probability that at least one of the $y$ bits in the frame signature is set to $0$ is 

$$
(\frac{y}{1})p(x, y, 1) - (\frac{y}{2})p(x, y, 2) + (\frac{y}{3})p(x, y, 3) - \ldots + (-1)^{y-1}(\frac{y}{y})p(x, y, y).
$$

The probability that a particular set of $y$ bits is set to 1's in a frame signature generated by superimposing $x$ bit strings, $P(x, y)$, is $1$ - the probability that at least one of the $y$ bits in the frame signature is set to 0. As a result,

$$
P(x, y) = 1 - \sum_{i=1}^{y} (-1)^{i-1} \binom{y}{i} p(x, y, i)
$$

$$
= \sum_{i=0}^{y} (-1)^{i} \binom{y}{i} \left(\frac{m-i}{w_b}\right)^{x} \left(\frac{m}{w_b}\right)^{-(w_b)}.
$$

(A.1)

Next, we use intermediate steps to simplify the above equation.

$$
\binom{m-i}{w_b} / \binom{m}{w_b} = \frac{(m-i)!/(m-i-w_b)!w_b!}{m!/(m-w_b)!w_b!}
$$

$$
= \frac{(m-i)!/(m-i-w_b)!}{m!/(m-w_b)!}
$$

$$
= \frac{m-i}{m} \cdot \frac{m-i-1}{m-1} \ldots \frac{m-i-w_b+1}{m-w_b+1}
$$

$$
= (1 - \frac{i}{m})(1 - \frac{i}{m-1}) \ldots (1 - \frac{i}{m-w_b+1})
$$

(A.2)

Because $(1 + \epsilon)^n = 1 + n\epsilon + O(\epsilon^2) \approx 1 + n\epsilon$ when $\epsilon \ll 1$,

Equation (A.2) \approx

$$
(1 - \frac{1}{m})^i(1 - \frac{1}{m-1})^i \ldots (1 - \frac{1}{m-w_b+1})^i
$$

$$
= \left(\frac{m-1}{m}, \frac{m-2}{m-1} \ldots \frac{m-w_b}{m-w_b+1}\right)^i
$$

$$
= \left(\frac{m-w_b}{m}\right)^i
$$

$$
= (1 - \frac{w_b}{m})^i
$$

Apply above formulae to (A.1). When $y \ll m$,

$$
P(x, y) \approx \sum_{i=0}^{y} (-1)^{i} \binom{y}{i} (1 - \frac{w_b}{m})^{ix}
$$
Therefore, \( P(s, 1) \) represents the probability that a particular bit position in the frame signature is set to 1, where \( s \) is the number of distinct key values in the information frame. There are \( m \) bits in a signature, so the average number of 1's in a frame signature is:

\[
w_f = mP(s, 1) = m(1 - (1 - \frac{w_b}{m})^s).
\]

Since \( w_b \ll m \), the average number of 0's set in a frame signature is:

\[
\bar{w}_f = m - w_f = m(1 - w_b/m)^s \\
\approx me^{-w_b/s/m}.
\]

\[\blacksquare\]
Lemma 6.2.4.2 Given the length of a signature in bits, \( m \), the number of bit strings superimposed into the signature, \( s \), and the number of 1's set in the bit strings, \( w_b \), the false drop probability for the signature is:

\[
P_f = \left(1 - \frac{\bar{w}_f}{m}\right)^{w_b}
\]

where \( \bar{w}_f \) is the average number of 0's in the signature.

Proof of Lemma 6.2.4.2: For a query with single key value, the false drops probability is:

\[
P_f = \text{Probability}[\alpha_1 = 0 \land \alpha_2 = 0 \land \cdots \land \alpha_{\bar{w}_f} = 0]
\]

\[
= \frac{(m - \bar{w}_f)!/w_b!(m - \bar{w}_f - w_b)!}{m!/w_b!(m - w_b)!}
\]

\[
= \frac{(m - w_b)!/w_b!(m - \bar{w}_f - w_b)!}{m!/w_b!(m - \bar{w}_f)!}
\]

\[
= \frac{(m - w_b)!/(m - \bar{w}_f - w_b)!}{m!/(m - \bar{w}_f)!}
\]

\[
= \frac{m - w_b}{m} \cdot \frac{m - w_b - 1}{m - 1} \cdots \frac{m - w_b - \bar{w}_f + 1}{m - \bar{w}_f + 1}
\]

\[
= (1 - \frac{w_b}{m})(1 - \frac{w_b}{m - 1}) \cdots (1 - \frac{w_b}{m - \bar{w}_f + 1})
\]

\[
\approx (1 - \frac{1}{m})^{w_b}(1 - \frac{1}{m - 1})^{w_b} \cdots (1 - \frac{1}{m - \bar{w}_f + 1})^{w_b}
\]

\[
= \left(\frac{(m - 1)(m - 2)\cdots(m - \bar{w}_f)}{m(m - 1)\cdots(m - \bar{w}_f + 1)}\right)^{w_b}
\]
= \left( \frac{m - \bar{w}_f}{m} \right)^{w_b}

= \left( 1 - \frac{\bar{w}_f}{m} \right)^{w_b}
Bibliography


[29] W. Kim, "UniSQL/X Unified Relational and Object-Oriented Database System," 

the IEEE International Conference on Very Large Data Bases, Amsterdam, 1989, 
423–432.

[31] W. Kim, K.-C Kim & A. Dale, "Indexing Techniques for Object-Oriented Databases," 

[32] W. Kim et al., "Integrating an Object-Oriented Programming System with a 


[34] D.L. Lee, Y.M. Kim & G. Patel, "Efficient Signature File Methods for Text Re­
trieval.,” IEEE Transactions on Data and Knowledge Engineering, Vol. 7, No. 3, 

Oriented Database Systems,” Proceedings of Conference on Information and Knowl­
edge Management, Gathersberg, MD, Nov. 1994, 64–71.

Image Database," Journal of Visual Languages and Computing, Vol. 3, No. 4, 

Database Systems,” Proceedings of the 2nd International Computer Science Con­

[38] W.-C. Lee & D.L. Lee, "Short Cuts for Traversals in Object-Oriented Database 


