Vertical Class Partitioning and Complex Object Retrieval

in Object Oriented Databases

By

Chi-Wai FUNG, B.Sc., M.Sc.

A Thesis Presented to
The Hong Kong University of Science and Technology
in Partial Fulfillment
of the Requirements for
the Degree of Doctor of Philosophy
in Computer Science

Hong Kong, December 1998

Copyright© by Chi-Wai FUNG 1998
Authorisation

I hereby declare that I am the sole author of the thesis.

I authorise the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorise the Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.
Vertical Class Partitioning and Complex Object Retrieval

in Object Oriented Databases

By

Chi-Wai FUNG, B.Sc., M.Sc.

APPROVED:

K. KARLAPALEM, SUPERVISOR

Prof. Roland T. CHIN, HEAD OF DEPARTMENT

Department of Computer Science, HKUST

December 1998
Vertical Class Partitioning and Complex Object Retrieval

in Object Oriented Databases

By

Chi-Wai FUNG, B.Sc., M.Sc.

A Thesis Presented to
The Hong Kong University of Science and Technology
in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy in Computer Science

Hong Kong, December 1998

Abstract

Next generation database applications require extra modeling supports, higher performance requirements and new design techniques. Object oriented database (OODB) technology has been introduced to support the next generation database applications, as it provides rich modeling features like, encapsulation of methods, inheritance, object identity, arbitrary data types and complex objects. For efficient processing of the next generation database applications, however, there is a need to research techniques for performance oriented OODB design, because we can not simply adopt conventional database design techniques which do not fully consider physical characteristics of the schema, the occurrences of the underlying database data and the application workload.

Class partitioning is the process of clustering relevant data accessed by an application into a class. This is a relevant and important research topic for OODBs, because it allows to reduce the amount of irrelevant data accessed, thus reducing the number of disk accesses. In this thesis, the topic of vertical class partitioning in OODBs is thoroughly investigated through
an analytical approach, and a number of experiments are conducted to evaluate and demonstrate the "goodness" of this design technique. Our experiments show that there exists an optimal number of vertical fragments for a class collection, and vertical partitioning can give rise to substantial savings in the number of disk accesses. A Cost-Driven Algorithm (CDA) has been developed, which guarantees to produce the cost optimal partitioning scheme based on exhaustive enumeration, but it has a high computational complexity. We have therefore developed a Hill-Climbing Heuristic Algorithm (HCHA) based on both the cost-based and affinity-based approaches. This algorithm uses the initial solution generated by affinity-based algorithm and incrementally evolves it to generate mostly optimal or near optimal vertical class partitioning scheme.

As performance is a key factor for the success of OODB systems, efficient complex object retrieval in OODB systems has also become a relevant problem. This thesis further addresses the issue of complex object retrieval by introducing structural join index hierarchy (SJIH) mechanisms that mimic the class composition hierarchy of the complex objects to provide direct access to complex objects and their component objects. We show that SJIH can provide efficient and flexible access to complex objects, and also unify various previous indexing methods proposed for OODBs. Our analytical experimental results demonstrate the effectiveness of the SJIH mechanisms over pointer traversal and other indexing mechanisms, such as Multi-index and Nested index. An algorithm has been developed to select, for a given set of queries, optimal or near-optimal SJIH. Finally, we show how vertical class partitioning can be applied to further improve the effectiveness of SJIH mechanisms.
Dedication

To my wife Ada Chung
Acknowledgments

I would like to thank my supervisors Dr. Kamalakar Karlapalem and Dr. Qing Li for their guidance, support and encouragement while I was at HKUST. This thesis comes from numerous inspiring discussion with them. I consider myself to be very fortunate to have them as my supervisors.

I would like to thank Prof. Frederick H. Lochovsky, Dr. Dik Lee, Dr. Leon Zhao and Prof. Elisa Bertino for being my examiners. Their comments and suggestions help to improve my thesis greatly.

I would also like to thank for their help, encouragement and valuable discussion from my fellow students in the database group, Dickson Chiu, Ladjel Bellatreche, Patrick Hung and Tony Wong. I would like to express my appreciation for help from the staff of the Computer Science Department of HKUST, especially Connie and Isaac.

Finally, my heartfelt thanks go to my mother for her love and support. I am grateful to my dear Ada for her love and patience during my PhD study.
Table of Contents

Title Page ................................................................. i
Authorisation Page .................................................... ii
Signature Page ........................................................... iii
Abstract ................................................................ iv
Dedication ................................................................ vi
Acknowledgments ........................................................ vii
Table of Contents ........................................................ viii
List of Tables .............................................................. xiii
List of Figures ............................................................ xiv

1 Introduction ............................................................... 1
  1.1 Vertical Class Partitioning in OODBs ........................... 2
     1.1.1 Motivating example ........................................... 3
  1.2 Method-Induced Vertical Class Partitioning ............... 4
     1.2.1 Motivation for Method-Induced Vertical Class Partitioning 5
  1.3 SJIH: Structural Join Index Hierarchy ..................... 5
     1.3.1 Motivation for SJIH ........................................... 6
  1.4 Contributions of This Thesis .................................. 6
     1.4.1 Vertical Class Partitioning ............................... 6
     1.4.2 Method Induced Vertical Class Partitioning .......... 7
     1.4.3 Structural Join Index Hierarchy ....................... 7
  1.5 Applicability and Limitations ................................ 7
  1.6 Thesis Organization .............................................. 8
5.1.1 Basic definitions .................................................. 98
5.1.2 Representation of OODB schema with multiple paths .......... 98
5.1.3 Definitions of the different indices in SJIH .................... 99
5.1.4 Relationships among different indexing methods ............... 103
5.1.5 Types of Structural Join Index Hierarchy ...................... 104

5.2 Query Processing and Cost Model .................................. 105
5.2.1 Query processing strategy ...................................... 105
5.2.2 Cost Model Parameters ......................................... 108
5.2.3 Storage cost ..................................................... 109
5.2.4 Index retrieval cost .............................................. 119
5.2.5 Index maintenance cost .......................................... 121
5.2.6 Number of Possible SJIHs ...................................... 125
5.2.7 Remarks ......................................................... 125

5.3 Evaluation of SJIH Indexing Methods ............................. 126
5.3.1 Comparison of SJIH against Pointer Traversal ................. 126
5.3.2 Comparison of Different Indexing Methods ..................... 131
5.3.3 Comparison of Index retrieval cost against Update cost ....... 135
5.3.4 Comparison of Different SJIHs in a Large OODB Schema ...... 140
5.3.5 Summary of results .............................................. 150

5.4 Discussion and Guidelines ......................................... 150
5.4.1 Comparison of different SJIHs for complex object retrieval .. 150
5.4.2 Guidelines for index designers .................................. 151

5.5 Summary .......................................................... 151

6 Optimal SJIH Algorithm .............................................. 153
6.1 Hill-Climbing Heuristic SJIH Algorithm (HCHSA) ............... 153
6.1.1 HCHSA ...................................................... 154
6.1.2 Illustrative example ............................................. 154
6.1.3 Evaluation of the Effectiveness of HCHSA .................... 156
6.1.4 Index selection with limited storage space .................... 158

6.2 Experiment with Random Queries ................................. 159
6.2.1 Experimental Setup ............................................. 159
6.2.2 Results ....................................................... 160

6.3 Experiment on HCHSA Performance Gain .......................... 161
6.3.1 Experimental Setup .................................................. 162
6.3.2 Results ................................................................. 162
6.4 Integration of VCP Techniques into SJIH ............................. 164
6.5 Summary ................................................................. 168

7 Conclusions and Future Work ............................................. 169
  7.1 Summary of Research Contributions ................................. 169
    7.1.1 Vertical Class Partitioning .................................... 169
    7.1.2 Structural Join Index Hierarchy ............................... 170
  7.2 Implementation of VCP and SJIH in Commercial OODBMS ......... 171
  7.3 Future Work .......................................................... 171

References ............................................................................. 174

Appendix A  [64] Graph-based algorithm ................................. 181
Appendix B Derivations of elementary parameters ....................... 182
Appendix C Dependency of Index retrieval cost on Selectivity ......... 184
List of Tables

1. Method Execution Cost Terms ................................................................. 17
2. Cost Model Parameters for VCP .............................................................. 28
3. Component-wise cost model formulae for LMH case ................................. 33
4. Component-wise cost model formulae for SMH case .................................. 34
5. Query Characteristics .............................................................................. 64
6. Comparison of MSE and CVP experiment results ..................................... 71
7. Results on Analysis of Query Execution Cost ........................................ 72
8. Cost model parameters for MI-VCP ......................................................... 80
9. Cost model formulae for MI-VCP ............................................................ 84
10. Complex Object Retrieval Queries .......................................................... 105
11. Cost model parameters for SJH .............................................................. 109
12. Cost model parameter values used .......................................................... 109
13. Cardinality and forward fan-out values ................................................ 115
14. Number of SJH tuples for Different Cases ............................................ 115
15. K and OP ............................................................................................... 117
16. Best SJH for different cases .................................................................... 151
17. Query Access Patterns .......................................................................... 166
18. Results for Index retrieval cost .............................................................. 167
19. Results for Normalized IO and Storage ................................................ 167
List of Figures

1. Vertical class fragments V1, V2, V3 and composite object E* of class Employee .......... 4
2. Taxonomy of OODB indexing methods ........................................................................ 17
3. A CAD Application Schema ........................................................................................ 19
4. Example Occurrence of a Complex Design Object ......................................................... 20
5. Multi-index from class AtomicPart to CompositePart ................................................... 20
6. Multi-index from class CompositePart to AtomicPart ................................................... 21
7. Join Index between class AtomicPart and CompositePart ............................................ 21
8. Nested Index from class Connection and CompositePart .............................................. 21
9. Path Index from class AtomicPart to classes CompositePart and Design ....................... 22
10. Access Support Relation between the classes AtomicPart, CompositePart and Design .......... 22
11. Object-based Class Approach of internal representation of vertical class fragments........... 26
12. Embedded Class Approach of internal representation of vertical class fragments ............ 26
13. Schema for a path expression in vertically partitioned case ........................................... 35
14. NIO vs. Number of Fragments (for number of object per page = 1/16, fan-out = 2.0, projection ratio = 0.2) ................................................................................................................... 45
15. NIO vs. Number of Fragments (for number of object per page = 16, fan-out = 2.0, projection ratio = 0.2) ................................................................................................................... 46
16. NIO vs. Number of Fragments for different scan strategies (for number of object per page = 1/16, fan-out = 2.0, projection ratio = 0.2) ........................................................................ 47
17. NIO vs. Number of Fragments for different scan strategies (for number of object per page = 16, fan-out = 2.0, projection ratio = 0.2) ........................................................................ 47
18. NIO vs. Fan-outs ............................................................................................................. 49
19. NIO vs. Cardinalities ..................................................................................................... 49
20. NIO vs. Projection Ratio (for object size 128KB) ............................................................ 51
21. NIO vs. Projection Ratio (for object size 0.5KB) .......................................................... 51
22. 3D plot of Projection Ratio vs. Object Size vs. Selectivity (for number of fragments = 8, fan-out = 2.0) .............................................................................. 52
23. 3D plot of Projection Ratio vs. Object Size vs. Selectivity (for number of fragments = 32, fan-out = 2.0) .............................................................................. 53
24. 3D plot of Projection Ratio vs. Object Size vs. Selectivity (for number of fragments = 8, fan-out = 0.125) .............................................................................. 53
25. 3D plot of Projection Ratio vs. Object Size vs. Selectivity (for number of fragments = 32, fan-out = 0.125) .............................................................................. 54
26. Min Square Error vs. Number of Fragments ........................................................................ 67
27. Scatter Plot of Square Error vs. Number of Fragments ....................................................... 68
28. Min Total IO Cost vs. Number of Fragments ..................................................................... 69
29. Scatter Plot of Total IO Cost vs. Number of Fragments ..................................................... 70
30. MDG of an example method ma() ...................................................................................... 76
31. MDG, sample code and cost formula for (a) simple method invocation, (b) conditional method invocation, and (c) repeated method invocation .................................................. 77
32. Example schema for a MDG path in Unpartitioned case .................................................... 81
33. Example schema for a MDG path in Vertical partitioned case ............................................. 82
34. Cost-Driven Algorithm ....................................................................................................... 86
35. (a) Instance variable usage matrix (IVUM) and (b) Instance variable affinity matrix (IVAM) .............................................................................................................. 87
36. Hill-Climbing Heuristic Algorithm ...................................................................................... 90
37. MDG of complex method expense() ................................................................................ 92
38. Normalized IO for case (i) of even distribution of instance variable accesses ............... 94
39. Normalized IO for case (ii) of skewed distribution of instance variable accesses ......... 94
40. Class Composition Hierarchy ............................................................................................ 99
41. A CAD Application Schema ............................................................................................. 100
42. Example Occurrence of a Complex Design Object .......................................................... 100
43. BJI between classes AtomicPart and CompositePart ...................................................... 100
44. DII between classes Connection and CompositePart ....................................................... 101
45. HJI between classes Documentation and Engineer ........................................................ 103
46. SJI between classes Design, Engineer, Office, Documentation and AtomicPart ............ 103
47. Relationships among the different indexing methods ....................................................... 104
77. For Scale factor of 0.5 and 1,000,000 Updates ................................................. 140
78. For Scale factor of 4.0 and 1,000,000 Updates .................................................. 140
79. Large Database Schema with 40 classes ......................................................... 141
80. PSJIH1 .............................................................................................................. 142
81. PSJIH2 .............................................................................................................. 142
82. PSJIH3 .............................................................................................................. 142
83. PSJIH4 .............................................................................................................. 142
84. Scale factor vs. Storage cost ............................................................................... 144
85. Scale factor vs. Cost ratio for Q2 (accessing 2 randomly selected classes) ............. 146
86. Scale factor vs. Cost ratio for Q5 (accessing 5 randomly selected classes within the same subtree) ........................................................................................................ 146
87. Scale factor vs. Cost ratio for Q10 (accessing 10 randomly selected classes within the same subtree) ........................................................................................................ 146
88. Scale factor vs. Cost ratio for Q20 (accessing 20 randomly selected classes within the same subtree) ........................................................................................................ 146
89. Scale factor vs. Cost ratio for Q30 (accessing 30 randomly selected classes within the same subtree) ........................................................................................................ 147
90. Scale factor vs. Cost ratio for Q40 (accessing all 40 classes) .................................. 147
91. Scale factor vs. Cost ratio for BSJIH ..................................................................... 149
92. Scale factor vs. Cost ratio for PSJIH1 ................................................................... 149
93. Scale factor vs. Cost ratio for PSJIH2 ................................................................... 149
94. Scale factor vs. Cost ratio for PSJIH3 ................................................................... 149
95. Scale factor vs. Cost ratio for PSJIH4 ................................................................... 150
96. Scale factor vs. Cost ratio for CSJIH ................................................................... 150
97. Hill-Climbing Heuristic SJH Algorithm ................................................................. 154
98. Example schema for HCHSA ............................................................................. 154
99. Query 1 .............................................................................................................. 155
100. Query 2 ........................................................................................................... 155
101. Query Graph .................................................................................................... 155
102. SQG after branch separation ............................................................................ 155
103. Initial SJH for the HCHSA .............................................................................. 155
104. Optimal SJH for both EEA and HCHSA .......................................................... 155
105. Scatter plot of Storage space vs. Total cost for all SJHs probed during the EEA. ... 159
106. Scatter plot of Storage space vs. Total cost for all SJIHs probed during the HCHSA159
107. Plot of Maximum number of SJIs vs. Final Cost/Initial Cost ........................................ 162
108. Plot of Maximum number of SJIs vs. No of Steps ......................................................... 163
109. Complete-SJIH .............................................................................................................. 166
110. Vertically Partitioned Complete-SJIH ............................................................................. 166
Chapter 1

Introduction

Database applications have evolved from simple applications (manipulating structured records containing numbers and character data) towards complex applications (accessing complex data, e.g., hypermedia data containing audio, video and images). These next generation database applications require extra modeling support, higher performance and new design techniques. Examples are multimedia systems (with text, audio, video and compound documents), web-based systems and engineering design (CAD/CAM) systems.

Object Oriented Database (OODB) products have been around as early as 1987. OODB technology has been introduced to support the next generation database applications [44,11,3]. It provides rich modeling features like, encapsulation of methods, inheritance, object identity, arbitrary data types and complex objects. It provides support for management of multimedia data, long-duration transactions and versioning.

Conventional database design techniques concentrate on conceptual and logical design, and are not performance oriented. For the relational database, Entity-Relationship model [16] is used for conceptual database modeling [2]. For the OODB, commonly used object oriented analysis and design methodologies include OMT [70], OOA/OOD [23,25], OOD [8] and UML [10,41,71]. All these design techniques do not consider physical characteristics of the schema and the occurrences of the underlying OODB, and the application workload. For efficient processing of the next generation database applications, there is a need to research techniques for performance oriented OODB design.

Class partitioning is the process of clustering relevant data accessed by an application into a class. This reduces the amount of irrelevant data accessed, thus reducing the number of disk accesses. Database design through class partitioning enhances the performance of the applications.
As emerging OODB applications become more and more complex, performance becomes a critical issue. Therefore, class partitioning in OODBs is a relevant and important research topic. One such application area for class partitioning is multimedia/video object database management system for which performance is a critical factor for successful implementation [57]. The number of alternative vertical class partitioning schemes for a given class is of the order of $n^n$ where $n$ is the number of instance variables. Therefore, there is a need to develop algorithms that are not only efficient but also provide good quality solutions. In order to evaluate the quality of a solution, the number of disk accesses needed to process a given set of queries/applications is determined. In this thesis, we research the topic of vertical class partitioning by evaluating the effectiveness of vertical class partitioning in OODBs for both queries and methods, and developing algorithms to come up with optimal or near-optimal vertical class partitioning schemes.

Performance is a key factor for the success of OODB systems, especially those supporting semantically rich database applications such as engineering applications, wherein complex object retrieval has a major impact on the cost of processing the queries. Thus, efficient complex object retrieval in OODB systems has become a problem of paramount importance. We introduce the structural join index hierarchy (SJIH) mechanism that mimics the class composition hierarchy of the complex objects to provide direct access to complex objects and their component objects. Structural join index hierarchy uses the work on join index hierarchies to provide efficient and flexible access to complex objects, thus unifying various previous indexing methods proposed for OODBs. In this thesis, we not only evaluate the "goodness" of different SJIHs, but also develop algorithm to select a set of SJIHs for a given set of queries. Finally, we show how vertical partitioning can be employed to further reduce storage costs and improve the SJIHs.

1.1 Vertical Class Partitioning in OODBs

There have been essentially two kinds of database design techniques. These database design techniques such as conceptual schema design, view integration and normalization, aim at capturing appropriate integrity constraints and complete incorporation of database application requirements. Whereas other techniques, such as index selection and database clustering, aim towards efficiently executing next generation database applications. Vertical partitioning is a technique for facilitating efficient execution of next generation database applications by reducing irrelevant instance variable (attribute) access. Though this problem has been addressed in relational database systems, there is very little work done on vertical partitioning in object oriented database (OOD) systems. For OODBs, new techniques are needed to facilitate effective partitioning, because traditional relational approaches to these problems are inadequate due to a number of unique features of object-orientation: complex object, encapsulation and inheritance. In particular, re-use of partitioning
techniques from relational databases in OODB environment has several drawbacks [48,50,49], as these techniques do not capture the behavioral/method semantics, maintenance of fragmentation transparency is difficult and costly, neither do they support encapsulation.

In an OODB context, vertical class partitioning (VCP) is a design technique for reducing the number of disk accesses required to execute a given set of queries by minimizing the number of irrelevant instance variables accessed. This is accomplished by grouping the frequently accessed instance variables as vertical class fragments as explained below.

1.1.1 Motivating example

We start with a motivating example. Consider the following schema for class Employee:

```java
Class Employee {
    EmpId char[8];
    Name char[40];
    Title char[20];
    Tel char[12];
    Fax char[12];
    Street char[40];
    City char[20];
    Country char[20];
    Zip char[10];
    Qualification char[200];
    Experience char[200];
}
```

The size of an Employee object is 582 bytes. Let class Employee be vertically partitioned into 3 fragments: V1, V2 and V3, with fragment V1 containing the instance variables EmpId, Name, Title, Tel and Fax, fragment V2 containing the instance variables Street, City, Country and Zip, and fragment V3 containing the instance variables Qualification and Experience. If most of the accesses to Employee class are for the instance variables of fragment V1, then the cost ratio of accessing only fragment V1 to that of the whole unpartitioned object is 92/582=16%, implying a cost saving of 84%, which is quite substantial\(^1\).

Figure 1 shows the internal representation of class Employee with vertical partitioning scheme \(\{V1, V2, V3\}\). The original class Employee can be internally represented by a class \(E^*\) with a set of object-based instance variables: \(io1, io2\) and \(io3\), where each object-based instance variable refers to an object from vertical fragment \(V_j\) (for \(1 \leq j \leq 3\)). This embodies an object oriented representation of a vertical partitioning of a class wherein each of the vertical class fragments is represented as a class, and a logical object of class Employee is internally represented as a composite object (of class \(E^*\)) that is composed of objects from the vertical class fragments. As vertical class

\(^1\) This is only a rough estimation using the storage space, Chapter 3 will present discussion that take into account the effect of buffering.
fragments are also represented as classes, this approach of representing vertical class fragments has
the added advantage of supporting fragmentation transparency. The usefulness of vertical parti-
tioning is two fold: (a) from a performance point of view, it reduces irrelevant data accesses by
grouping frequently accessed instance variables together to form vertical fragments; and (b) from
a design/semantic point of view, a vertical fragment (say \( V_I \)) is a component object of a more com-
plex object (\( Employee \)).

![Diagram](image)

**Figure 1:** Vertical class fragments \( V_1, V_2, V_3 \) and composite object \( E^* \) of class \( Employee \)

### 1.2 Method-Induced Vertical Class Partitioning

In an OODB, methods encapsulated in a class typically access a few, but not all the instance vari-
ables of the class. It may thus be preferable to vertically partition the class for reducing irrelevant
data (instance variables) accessed by the methods. Prior work has shown that vertical class parti-
tioning can result in a substantial decrease in the total number of disk accesses incurred for executing a set of applications, but coming up with an optimal vertical class partitioning scheme is an NP-
complete problem [68]. In this thesis, we present two algorithms for deriving optimal or near-opti-
timal vertical class partitioning schemes. The cost-driven algorithm (CDA) provides the optimal
vertical class partitioning schemes by enumerating, exhaustively, all the schemes and calculating
the number of disk accesses required to execute a set of applications. For this, a cost model for ex-
ecuting a set of methods in an OODB system is developed. Since exhaustive enumeration is costly
and only works for classes with a small numbers of instance variables, a hill-climbing heuristic al-
algorithm (HCHA) is therefore developed, which takes the solution provided by an affinity-based algorithm and improves it by further reducing the total number of disk accesses incurred. We show that the HCHA algorithm provides a reasonable near-optimal vertical class partitioning scheme for executing a given set of methods.

1.2.1 Motivation for Method-Induced Vertical Class Partitioning

- In OODB systems, methods encapsulated in a class typically access a few, but not all the instance variables defined in that class. Therefore, a judicious vertical class partitioning of the class can drastically reduce the irrelevant instance variables accessed by these methods and improve performance. Most OODB query languages allow for method invocation in the project and select clauses, and only some, but not all, instance variables are accessed.
- The encapsulation feature of OODB systems, even though it complicates the partitioning problem, it has been utilized to provide fragmentation transparency support [50,49], thus enabling easier development of applications and their maintenance.
- The problem of coming up with an optimal vertical class partitioning scheme is NP-complete [68], and hence heuristic algorithms need to be designed to come up with near-optimal vertical class partitioning schemes. Two main heuristics, namely, affinity based [62] and cost driven [22] have been proposed in prior research.
- With the presence of methods in OODB systems, the cost driven approach requires the development of a cost model for method execution. This is a hard and relevant problem by itself, and is addressed in this thesis.

1.3 SJIH: Structural Join Index Hierarchy

For semantically rich database applications (such as engineering applications, and office automation), complex object retrieval has a major impact on the cost of processing the queries. Without any index or access support structures, complex object retrieval is processed by pointer chasing. But the cost of query processing using pointer traversal is very expensive, especially when: (1) the objects are large; (2) component objects to be retrieved are deep inside the class composition hierarchy; (3) traversing in the reverse direction of the path expression is required (due to the absence of reverse pointers); and (4) structural information (the links between the objects) needs to be retrieved.

To overcome the problems of expensive pointer traversal, many techniques have been proposed to expedite the query processing. These include indexing [7,51,76], function materialization [46], complex object assembly [43], and clustering [26]. In this thesis, we develop the concepts, and formulate the framework of the Structural Join Index Hierarchy (SJIH) that encompasses the
existing OODB indexing methods including Nested index, Path index [7], Access Support Relation [51] and Join Index Hierarchy [76].

1.3.1 Motivation for SJIH

- Complex object retrieval requires a large number of disk accesses using pointer traversal. In contrast, our framework of SJIH is aimed at facilitating direct access to complex objects. Furthermore, by mimicking the class composition hierarchy of complex objects, SJIH is targeted at providing direct access to (component) objects spreading over multiple paths.

- Most of the existing indexing techniques support either forward or reverse traversal, but do not provide a comprehensive framework for complex object retrieval in both directions. SJIH facilitates bi-directional traversal, so that we can retrieve data objects from either the root class or the leaf classes.

- One of the difficulties in using existing OODB indexing techniques is the lack of a comprehensive framework in which their advantages and disadvantages can be evaluated. The SJIH framework subsumes the previous work in OODB indexing, in that many of the existing indexing techniques, such as Nested index and Join index hierarchy, are special cases of SJIH. This facilitates a comprehensive evaluation of different indexing schemes, and development of further work towards the SJIH selection problem.

1.4 Contributions of This Thesis

1.4.1 Vertical Class Partitioning

The main contributions of this thesis on Vertical Class Partitioning include:

- Development of a comprehensive analytical cost model for query processing on vertically partitioned classes that encompasses both class composition hierarchy and class inheritance hierarchy.

- Detailed discussion of the extra overhead of processing composite objects due to vertical partitioning, by showing that this overhead is relatively small if the number of composite objects is small. This overhead will, however, blow up if the objects in a class collection are vertically partitioned into excessively many small vertical fragments.

- Demonstration of the fact that the projection ratio is the most influential factor in deciding whether a given vertical partitioning scheme is beneficial or not.

- Illustration of the complementary nature of indexing to vertical partitioning, in that the former can be used (on the root class) alongside with the latter; furthermore, there is a large range of parameter values within which vertical partitioning is superior to indexing.
1.4.2 Method Induced Vertical Class Partitioning

The main contributions of this thesis on Method-Induced VCP are as follows:

- We develop a cost model for method execution in both unpartitioned and vertically partitioned OODBs. This cost model is used in the subsequent algorithms to generate the optimal vertical class partitioning scheme. To the best of our knowledge, this is the first piece of work on developing a cost model for complex method execution in OODBs.

- We develop a Cost-Driven Algorithm (CDA) which guarantees to produce the cost optimal partitioning scheme based on exhaustive enumeration, which has a high computational complexity $O(n^n)$, where $n$ is the number of instance variables in the class.

- We present an affinity-based algorithm which has low computational complexity of $O(n^2)$. We show that the affinity-based algorithm does not necessarily generate the optimal vertical class partitioning scheme.

- Finally, we develop a Hill-Climbing Heuristic Algorithm (HCHA) based on both the cost-based and affinity-based approaches. This algorithm uses the initial solution generated by the affinity-based algorithm and incrementally evolves it to generate mostly optimal or near optimal vertical class partitioning schemes.

1.4.3 Structural Join Index Hierarchy

The main contributions of this thesis on Structural Join Index Hierarchy include:

- We formulate a framework of SJIH which subsumes previous work in OODB indexing methods and yet provides efficient support for complex object retrieval.

- We develop a detailed cost model for the retrieval cost, the storage cost and the index maintenance cost for SJIH.

- We perform analytical experiments to show that there is a wide range of parameters for which SJIH is beneficial.

- We present and conduct experiments on the SJIH algorithm for optimal index selection. The results show that the SJIH algorithm is both efficient and effective in finding the optimal or near-optimal SJIH.

- We show how vertical class partitioning techniques can be employed to further reduce storage and retrieval costs for SJIH.

1.5 Applicability and Limitations

The aim of vertical class partitioning is to reduce the amount of irrelevant data accessed by an application. This is especially useful for the next generation database applications, such as document management, multimedia and hypermedia systems, in which many of the instance variables tend
to be very large objects that should not be accessed if they are not actually needed by the applications. Although (in later chapters) we illustrate VCP using a centralized OODB environment, by extending the cost model and algorithm, it is also applicable to a distributed OODB environment. Structural join hierarchy (SJIH) mechanisms mimic the class composition hierarchy of the complex objects to provide direct access to complex objects and their component objects. This is especially useful for semantically rich applications such as engineering design applications. Although we illustrate our performance-oriented techniques as applied to OODBs, these techniques are equally applicable for object-relational databases, as both OODBs and object-relational databases utilize the same object oriented technology.

Support for class partitioning in current commercial OODBs is still in its infancy. But with the results showing the utility of vertical class partitioning, it is possible that vertical class partitioning techniques will be supported by OODBs in future. Even if this is not the case, for specific applications, vertical class partitioning can be hard-coded to provide efficient execution of the applications. Complex object retrieval is a feature that is supported by most commercial database systems, by using different indexing schemes. Therefore, it is possible that commercial OODBs will support SJIH for complex object retrieval.

The results show that both vertical class partitioning and complex object retrieval techniques require judicious incorporation based on the database system parameters, database characteristics, and application characteristics. Thus, if these characteristics cannot be accurately determined, then the applicability of the proposed techniques is diminished. But, on the whole, general guidelines are provided to compare the utility of these techniques to different database and application characteristics.

1.6 Thesis Organization

The remaining chapters of the thesis are organized as follows: Chapter 2 reviews previous work on VCP and conventional OODB indexing methods. Chapter 3 details our work on VCP. Chapter 4 presents our research results on method-induced VCP. Chapter 5 introduces our SJIH framework. Chapter 6 presents SJIH algorithms to select the optimal or near-optimal indexing scheme. Finally, Chapter 7 presents conclusions and future work.
Chapter 2

Previous Work

This chapter discusses previous work relevant to this thesis. Section 2.1 presents a core object oriented data model to facilitate our subsequent discussion. Section 2.2 reviews the previous work on partitioning in relational databases. Section 2.3 presents the status of research in class partitioning. Section 2.4 reviews previous work on vertical class partitioning. Section 2.5 presents related work on method execution cost models and some background work on method-induced vertical class partitioning. Finally, Section 2.6 discusses the previous work in OODB indexing methods.

2.1 A Core Object Oriented Data Model

In this section, we briefly review a collection of concepts that are essential to an OODB. To make our discussion general-purpose, we focus on the basic concepts that are mandatory and/or common to most OODB models and systems [47, 1, 44]. These elementary concepts also form the core of the OODB model that we shall assume for our subsequent discussion.

The most fundamental concepts of an OODB include class, inheritance, and object identifier (OID). A class defines a set of instance variables (or attributes) that constitute the "state", and a set of procedural methods that embody the "behavior" of its objects. Classes are organized into an inheritance (Is-A) hierarchy, in which a subclass inherits the attributes and methods defined in the superclass(es) for its objects. For each attribute, the set of values it may have is confined by its class type (which is called the domain class); both atomic (e.g., integer, string of characters) and abstract (i.e., object) domain classes are possible in an object. Objects are uniquely distinguished with their (system-generated) object identifiers (OIDs), hence the existence of an object is independent of its state (i.e., attribute values). Besides the Is-A hierarchy, OODBs tend to exhibit another form of useful hierarchy called the composition hierarchy, which captures the "Is-Part-Of" relation-
ship between a parent class and its component classes. A composite object $O$ can be defined as an object with a set of abstract attributes (whose domain classes are non-atomic), each of which refers to one or more component objects (or called subobjects) of $O$; the hierarchy of classes (to which $O$ and its subobjects belong) is called a class composition hierarchy. Composite objects add to the integrity features of an OODB model through the concept of existence dependency (i.e., whether the existence of the subobjects are dependent on the existence of the composite object), and the notion of shareability (which concerns whether the subobjects can be shared by several composite objects, or can only be used by one composite object exclusively). Thus there are four possible kinds of composition links (between the subobjects and their composite object): dependent and sharable, dependent and exclusive, independent and sharable, and independent and exclusive [42].

A method has a signature (interface) including the method’s name, a list of parameters, and a list of return values. Parameters and return values may be either value-based or object-based instance variables (VBIVs or OBIVs). Methods are inherited from the superclass(es). A subclass may alter the method code of an inherited method or additional methods may be added.

A simple method does not call/invoke any other method. A complex method calls/invokes other methods. A method that accesses VBIVs accesses the leaf node of a composition hierarchy. Therefore, a method that accesses OBIVs can potentially invoke other methods. A method can return atomic values (such as real, integers and string) or can return object identifiers.

2.2 Partitioning in relational databases

In conventional partitioning/fragmentation in relational databases, the most basic types of partitioning are: vertical partitioning and horizontal partitioning [63]. In vertical partitioning, a relation is divided into subrelations called vertical fragments which are projections of the relation on the subsets of original attributes. In horizontal partitioning, a relation is divided into subsets of tuples through selection operations, called horizontal fragments.

Many partitioning algorithms have been proposed for relational databases, which can be classified into: affinity-based algorithms and cost-based algorithms. In affinity-based algorithms, either attribute affinity (for vertical partitioning) or predicate affinity (for horizontal partitioning) is used as a guideline for performing the partitioning. In cost-based algorithms, an analytical cost model is used to calculate and compare the cost of different partitioning schemes.

As the number of attributes (for vertical partitioning) or the number of predicates (for horizontal partitioning) increase, the complexity (exponential) of finding the optimal partitioning scheme increases for exhaustive enumeration of all possible partitioning schemes. Therefore, we need to find good heuristics to tackle the problem of finding optimal partitioning scheme.
2.2.1 Vertical partitioning

The aim of vertical partitioning is to partition a relation into fragments, such that the optimal performance of the database system can be achieved. Selecting an optimal partition is a difficult problem. As pointed out by [38], a relation with $n$ attributes can be partitioned into $B(n)$ different ways, where $B(n)$ is the Bell number; for large $n$, $B(n)$ approaches $n^n$. Thus heuristic approaches are necessary to determine the optimal or near-optimal partition for a relation with a large number of attributes.

Many of the previously proposed vertical partitioning algorithms are affinity-based. In an early work by Hoffer and Severance [39] the concept of attribute affinity was developed. Attribute affinity described the degree to which pairs of attributes are accessed together, and hence, grouping attributes with high affinity will reduce access costs. In [39], clustering of attributes on the basis of their affinity was achieved by applying the Bond Energy Algorithm (BEA) developed by McCormick [59].

Navathe and Ceri [61] extended the work of Hoffer and Severance [39] to a two phase affinity-based vertical partitioning algorithm. An iterative binary partitioning method was used, based on first clustering the attributes and then applying empirical objective functions to perform the partitioning. The computational complexity of this algorithm is $O(n^2 \log(n))$.

Navathe and Ra [64] proposed an affinity-based graph theoretic algorithm for vertical partitioning. It has complexity $O(n^2)$ which is less than [61]. These affinity-based methods can find a feasible partition, but partitioning by attribute affinity does not imply that the partitioning scheme found can reduce the total number of disk accesses as elaborated by [22,24].

Chu and Leong [17], developed a vertical partitioning method that optimizes the number of disk accesses based on clustering of attributes accessed by transactions, then they compare their method with other methods.

In [18], an affinity-based objective function that generalizes and subsumes earlier work is derived. It makes use of some data clustering algorithms for distributed design. The results of different previously proposed vertical partitioning algorithms are sometimes different even for the same attribute affinity matrix, indicating that the objective functions used by these algorithms are different. Most of the proposed vertical partitioning algorithms do not have an objective function to evaluate the "goodness" of partitionings that they produced. Also, there is no common criterion or objective function to compare and evaluate the results of these vertical partitioning algorithms. The objective function derived in [18] is also used for developing heuristic algorithms.

Due to this difficulty in comparing the "goodness" of the resulting vertical partitioning schemes in affinity-based approach, cost-based partitioning algorithms are proposed. Cornell and
Yu [22,24] developed an analytical cost model that concentrates on the total number of disk accesses.

2.2.2 Horizontal partitioning
Ceri, Negri and Pelagatti [19] analyze the horizontal partitioning problem, which deals with the specification of partitioning predicates. Since each fragment is given by the conjunction of all the predicates in the natural form or negated form, the maximum possible number of fragments can be $2^n$ (where $n$ represents the number of predicates), which is quite a large number and with high computation complexity.

Most of the previous work on horizontal partitioning algorithms are affinity-based. Ceri, Navathe and Wiederhold [20] develop an optimization model for horizontal partitioning. The logical schema of a database is modeled as a directed graph with objects as nodes and links as edges, and required the user to specify the information about: (1) the data about the schema (like attribute size, relationship size and cardinality), (2) transaction characteristics (like frequencies and site of origin), (3) distribution requirement and/or constraints.

[69] presents an affinity-based graphical algorithm for horizontal partitioning. In this algorithm, horizontal fragments are determined by using the clustering of predicates that are based on the predicate affinity. For clustering, this algorithm adopts the graphical partitioning technique developed in his previous research for vertical partitioning. The graph-based algorithm generates a relatively small number of horizontal fragments and of lower computation complexity as compared with previous horizontal partitioning algorithms.

2.3 Status of Research in Class Partitioning
Partitioning is an important technique originally developed for reducing (data) access for record-oriented databases. The basic premise behind partitioning is to reduce irrelevant data access by grouping the attributes (for vertical partitioning) or the object instances (for horizontal partitioning) frequently accessed together as fragments. Though this problem has been addressed in relational database systems, there is very little work on class partitioning for object oriented database (OODB) systems. The reason for this is due to the complexity of OODB models primarily the class inheritance hierarchy and class composition hierarchy, which complicate the definition and representation of class fragments.

2.3.1 Vertical and horizontal class partitioning
There has been some work done in class partitioning algorithms in OODBs. In [27,28], the authors presented approaches to horizontal and vertical class partitioning of OODBs, which is based on the concept of affinity [62]. The approach taken there is to adopt the top down methodology, taking
into account such factors as application access patterns (queries), frequency of queries, object affinity, method and instance variable types, etc. However, there was no representation scheme provided for fragments. Also, the physical disk access costs corresponding to the savings on the amount of irrelevant data accessed and the overhead incurred due to partitioning were ignored. Further, their approach does not consider the issue of methods inducing different types of partitioning schemes.

In order to support OODB systems we need to have a clear understanding of what is meant by partitioning a class and what are the different ways of doing so. In [48,49], they concentrate on studying the different types of partitioning schemes that can arise in OODBs. They laid down a foundation by articulating the concepts, representation, and implementation approaches for different partitioning schemes for OODBs, and thus facilitate further work on partitioning algorithms.

[48,49] presented three major partitioning schemes for OODBs, namely, vertical class partitioning, path partitioning and horizontal class partitioning. Their approach in developing these partitioning schemes is to assure that all the resultant class fragments can be represented and implemented as classes in an OODB. This uniform representation facilitates the support of fragmentation transparency for class partitioning. As discussed in [48,49], although the concepts of vertical and horizontal partitioning resemble the vertical and horizontal partitioning in relational databases, additional complexities and implications arise in OODBs due to composite objects, class inheritance hierarchy and support for methods.

In designing each of the partitioning schemes [48,49], a fundamental issue that had to be considered was the role of "methods", since methods are the basic primitives for accessing the objects. Depending on the different types of methods, and the values returned by the methods, various combinations of the partitioning schemes can be developed resulting in what is known as method-induced partitioning schemes. A method can either be a simple method or a complex method. A simple method does not call/invoke any other method. In particular, if a simple method is in one-to-one correspondence with an instance variable and its sole purpose is to retrieve the value of that instance variable, we call it an elementary instance variable access method. A complex method calls/invokes other methods. In [50,49], a complex method is represented by method dependency graphs (MDGs). Each method dependency graph represents the behavior of a complex method accessing an object-based instance variable (OBIV). An MDG has a set of nodes and a set of directed edges. The nodes represent the methods invoked by the complex method, and the edges denote the partial order in which the methods were invoked. The edges are labeled with the OBIV that is being accessed next by the complex method. The method to be invoked next will be a method defined in the home class of the OBIV. Since a complex method can invoke other methods (both simple and complex), a complex method can be generally represented by a set of MDGs.
In [50,49], the concepts of method induced partitioning schemes in OODBs by understanding and classifying the object behavior embodied by the methods was presented. Further, a solution for supporting fragmentation transparency by using method transformation and guidelines for method induced partitioning for OODBs were developed.

2.3.2 Path partitioning
Path partitioning is a new form of partitioning in OODBs first introduced in [48,50,49]. For many applications, it is required to access the complete composite object [48,49]. Path partitioning is a concept describing the clustering of all the objects forming a composite object into a partition. A path partition consists of grouping the objects of all the domain classes that correspond to all the instance variables in the subtree rooted at the composite object. The concept of path partitioning is very useful in engineering design databases wherein a user accesses a complete composite object and does not want to traverse through multiple classes to extract all the relevant instance variables.

The path fragment can be internally represented as a "structural index" for the composite object. Each path partition can be represented as a hierarchy of nodes forming a structural index. Each node of the structural index points to the objects of the domain class of the component object. The instances of the structural index are a set of OIDs that point to all the component objects of a composite object.

Similar to path partitioning, the use of indexing (like path index, access support relation and path dictionary [7,51,52,55,56]) reduces the number of disk IOs in query execution for a particular path expression in the OODB by reducing the accesses to irrelevant object instances. But path partitioning is more general than indexing in that it support the access of the whole composite object, and is not restricted to one particular path expression. In this thesis, a representation scheme for path fragments, namely, structural join index hierarchy (SJIH) is developed.

As pointed out by [49], both vertical class partitioning and path partitioning are based on the intention of the OODB, whereas horizontal partitioning is based on the extension of the OODB. In this thesis, we concentrate on vertical class partitioning and path partitioning/SJIH.

2.4 Vertical Class Partitioning and Method Induced Vertical Class Partitioning

2.4.1 Partitioning versus Clustering, Indexing and Complex object assembly
In contrast to partitioning, indexing in OODBs is a facility to reduce the number of disk IOs in query execution (see, e.g., [7,51]) by reducing the accesses to irrelevant object instances (as compared with sequential scanning). Indexing reduces disk IOs at the object instance level; it still accesses irrelevant instance variables, since not all the instance variables accessed are relevant to the query.
Vertical partitioning is also a facility in OODBs to reduce the number of disk IOs for application execution by reducing the accesses to irrelevant instance variables. But unlike indexing, vertical partitioning reduces disk IOs at the instantiation level.

In [43], an "assembly" operator is devised to address the problem of avoiding "pointer chasing on disk". This assembly operator is designed to improve processing of bulk data types such as sets in object bases. When compared with vertical partitioning, complex object assembly concentrates on the physical object instance level, whereas vertical partitioning concentrates on the instance variable level. During query processing, complex object assembly still accesses irrelevant instance variables. If the irrelevant instance variables are large, complex object assembly may not be very useful.

We note that partitioning is a logical database design technique whereas clustering/indexing/complex object assembly is a physical database design technique. The use of indexing is complementary and orthogonal to the use of partitioning. That is, once the classes are vertically partitioned, indices can be built to execute the applications more efficiently. Since vertical class fragments redefine the OODB schema with new classes, clustering and complex object assembly can be applied to further improve efficient execution of applications. The issue of integrating clustering, indexing and complex object assembly with vertical partitioning is beyond the scope of this thesis.

2.5 Method Execution Cost Model

In most of the previous research, encapsulated methods in OODBs are treated as black boxes and are neglected from cost analysis. There has been very little discussion on the cost model for general method execution, due to the complexity of the detailed semantics of a general method. Unlike relational databases, query processing in OODBs have to deal with methods, whose different types of invocation affect the cost model. We consider three types of invocations found in general purpose programming languages: simple invocation, conditional invocation, and repeated invocation. To utilize this model, run-time invocation frequencies are collected through monitoring the past runs. The novelty of this approach is the use of methods of OODBs as input for describing the application load on each instance variable. Methods' semantics can be abstracted by means of graphs describing the flow of execution with control structures.

Related Work

In Object Oriented Databases (OODBs), access to data are through encapsulated methods. The encapsulation property of methods in OODB makes the estimation of execution cost more difficult than conventional query processing in relational databases. Furthermore, the encapsulation of methods also complicates the issue of query optimization. In some early studies of OODB query
optimization, methods are not considered in query optimization. However, some systems overcome this difficulty by treating the query optimizer as a special application that can break encapsulation and access information directly [13]. But in order to come up with optimal method execution plans, an analytical cost model for method execution is essential.

In this thesis, we present a cost model for general method execution. We observe that complex methods can be broken down into sub-methods. The method execution costs are dependent upon the types of invocation from a method to its sub-methods. These types of invocations include simple, conditional and repeated method invocations. The main contribution of this thesis is the development of an analytical cost model for general method execution in OODBs.

The following is some related work on method execution cost models in OODBs:

1. In [77], the optimization of queries containing methods is discussed. Some new strategies of object query optimization for the classification and evaluation of selections and joins containing path expressions or methods are proposed. Object query graphs are used to represent object queries and to capture different kinds of selections and joins. Object query graphs are also used to factorize common sub-path expressions among not only path expressions in queries but also maximal common sub-path expressions in methods and to generate query evaluation plans.

2. In [72], although they do not propose a specific cost model, they assume that the OODBMS is capable of using OID-stream statistics to derive a cost for method calls. Appropriate OID-stream statistics might be stream cardinality and information about the classes represented in the stream. For a given method call, the OODBMS could derive a processing cost and statistics for the resulting output OID-stream.

3. In [40], they developed a model for measuring the cost of a predicate function (similar to our notion of method). The cost of a predicate function is computed by adding up the costs for each sub-predicate function in the predicate function expression. Given a predicate function \( p(a_1, ..., a_n) \), the expense per object is recursively defined as:

\[
e_p = \begin{cases} 
\sum_{i=1}^{n} e_{a_i} + \text{percall_cpu}(p) + \\
\text{perbyte_cpu}(p) \times \left( \frac{\text{byte_pct}(p)}{100} \right) \times \sum_{i=1}^{n} \text{bytes}(a_i) + \text{access_cost} \\
0 & \text{if } p \text{ is a predicate function} \\
0 & \text{if } p \text{ is constant or instance variable}
\end{cases}
\]
Table 1: Method Execution Cost Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>percall_cpu</td>
<td>execution time per invocation, regardless of the size of the arguments</td>
</tr>
<tr>
<td>perbyte_cpu</td>
<td>execution time per byte of arguments</td>
</tr>
<tr>
<td>byte_pct</td>
<td>percentage of arguments bytes that the function will need to access</td>
</tr>
<tr>
<td>bytes</td>
<td>expected (return) size of the argument in bytes</td>
</tr>
<tr>
<td>access_cost</td>
<td>cost of retrieving any data necessary to complete the function</td>
</tr>
</tbody>
</table>

And $e_{a_i}$ is the recursively computed expense of argument $a_i$. The parameter values of the above table are based on initial estimation of either default values or system statistics. After repeated applications of a predicate function, one could collect performance statistics and use curve-fitting technique to make estimates about the predicate function's behavior.

2.6 OODB Indexing Methods

2.6.1 Taxonomy of OODB Indexing Methods

In this section, we present a taxonomy of OODB indexing methods. As illustrated in Figure 2, there are two main categories of OODB indexing methods: (a) index for class composition hierarchies, and (b) index for class inheritance hierarchies.

![Figure 2: Taxonomy of OODB indexing methods](image-url)
Index for Class Composition Hierarchies

Previous work on indices for class composition hierarchies includes:

- Multi-index (MI) [60]
- Join index (JI) [74]
- Nested index (NI) and Path index (PI) [7]
- Access Support relation (ASR) [51]
- Join index hierarchy (JIH) [76]

As shown in Figure 2, there are two types of indices for class composition hierarchies: namely variable length index—index with a list of pointers/OIDs in the index entry (that is, the size of the index entry can vary); and fixed length index—index with a fixed length tuple structure. Examples of variable length indices include MI, NI and PI. For variable length indices, in terms of the complexity of the index structure and the ease of index maintenance: MI is the simplest index and PI is the most complex index, with NI in between. Fixed length indices include JI, ASR and JIH. For fixed length indices, in terms of the complexity of the index structure and the ease of index maintenance: JI is the simplest index and both ASR and JIH are more complex than JI. JIH extends JI by building up a hierarchy of JIs. Further, we note that a JI only covers two classes, but ASR covers a number of classes along a path in the OODB schema, and this is the main difference between JI/JIH and ASR. Further discussion on previous work on indices for class composition hierarchies can be found in [4,12,5,9].

Index for Class Inheritance Hierarchies

Previous work on indices for class inheritance hierarchies includes:

- single-class index (SC-index) and class-hierarchy index (CH-tree) [45]
- H-tree [58]
- CG-tree [53]
- hcC-tree [73]

As shown in Figure 2, there are also two types of indices for class inheritance hierarchies: (i) indices that mainly cater for single-class queries (SC-queries), and (ii) indices that mainly cater for class-hierarchy queries (CH-queries). Examples of indices for SC-queries include SC-index, H-tree and CG-tree. An example of index for CH-queries is CH-tree. hcC-tree attempts to combine the advantage of both SC-index and CH-tree. Further discussion on previous work on indices for class inheritance hierarchies can be found in [15,9].

Finally, as shown in Figure 2 and we will explain in details in Chapter 5, the SJIH framework proposed in this thesis subsumes all the indexing methods for class composition hierarchies.
2.6.2 Example
Our subsequent discussion is based on the running example schema adapted from the OO7 Benchmark [14] as illustrated in Figure 3. The figure shows a schema of an example CAD application, which consists of a class composition hierarchy with four paths. A CAD design (Design) has a number of design teams (Team) and a number of composite parts (CompositePart). A team in turn can have a number of design engineers (Engineer) and a number of offices (Office). On the other hand, a composite part has a number of documentation (Documentation) to describe its design details and has a number of atomic parts (AtomicPart). Furthermore, an atomic part consists of a number of electrical connections (Connection).

2.6.3 Indexing Methods
There has been some work done on OODB indexing techniques for efficient query execution along a class composition hierarchy. These techniques include: Multi-index [60], Join index [74], Nested index [7], Path index [7], Access Support Relation [51] and Join index hierarchy [76]. We examine these indexing methods using the CAD example schema. In addition, we show in Figure 4 an occurrence of a complex Design object, in which only the OIDs are shown. We make use of the notation as in [7]: the D[], T[], E[], O[], C[], M[], A[] and N[] are the OIDs for the classes Design, Team, Engineer, Office, CompositePart, Documentation, AtomicPart and Connection, respectively. In particular, we let classes Engineer and CompositePart be constrained pair-up\(^1\), i.e., engineer E[1] is responsible for composite part C[1] only, but not for C[2].

![Figure 3: A CAD Application Schema](image)

---

1. Refer to Section 5.2.3 for detailed discussion on constrained pair-up.
A Multi-index (MI) allocates an index on two neighbouring classes [60]. Figure 5 shows an example MI from class AtomicPart to class CompositePart. It expedites the retrieval of OIDs of class CompositePart from any given OID of class AtomicPart. But for query processing, MI requires scanning the number of indices equal to the length of the path expression. Three MIs are needed to process a query that starts from class Connection and ends with class Design. Furthermore, an MI only supports traversal in one direction (usually from an ending class towards the root class in a path expression). This is because it is difficult to build another clustered index on a list of OIDs (cf. Figure 5, a list of OIDs of class CompositePart). Using the MI shown in Figure 5, given an OID of class CompositePart (say, C[1]) it is difficult to retrieve related AtomicPart OIDs. If we also want to traverse in the reverse direction, we need to build another MI clustered on the root class. An example of such an index is shown in Figure 6, where the MI is from class CompositePart to class AtomicPart, which expedites the retrieval of OIDs of class AtomicPart from any given OID of class CompositePart.
The Join index (JI) was originally introduced to efficiently perform joins in relational databases [74], and to expedite query processing in OODBs. Figure 7 shows an example JI between the classes AtomicPart and CompositePart. The JI supports both forward and reverse traversal (e.g., among classes AtomicPart and CompositePart in Figure 7). Both MI and JI require a number of indices equal to the path length to efficiently process the queries.

A Nested index (NI) [7] provides a direct association between objects of classes at the beginning and end of a path. It eliminates accesses to irrelevant intermediate classes. Figure 8 shows an example NI from class Connection to class CompositePart. Similar to MI, NI only supports unidirectional traversal.

A Path index (PI) is similar to a NI and is based on a single index concept [7]. The difference is that a PI considers the whole path. Unlike NIs, PIs can be used to expedite query processing in cases where the predicates are on classes along the path expression. Figure 9 shows an example PI on the sub-path Design.CompositePart.AtomicPart. PIs support uni-directional traversal like NI.

An Access Support Relation (ASR) in [51] is similar to a PI in that it involves all the object instances along a path expression. The main difference is that it stores the OIDs in the form of a relation. Figure 10 shows an example ASR along the sub-path Design.CompositePart.AtomicPart.
ASR supports traversal in multiple directions like JIs. This is accomplished by building up non-clustered indices on the ASR. For example, building non-clustered index on CompositePart OIDs facilitates the traversal from class CompositePart towards classes Design and AtomicPart.

A Join Index Hierarchy (JIH) method in [76] extends the JI concept from relational databases. JIH constructs a hierarchy of join indices and transforms a sequence of pointer traversal operations into simple search in an appropriate Join index. JIH supports traversal in both directions, but JIH supports only single path queries.

By comparing the above mentioned indexing methods, we can note the following points:

- In terms of the number of index lookups required for query execution, JIH, PI, ASR and NI are better than MI and JI.
- For cases with a low degree of sharing between composite objects and their component objects, JIH, PI and ASR are the best in terms of storage overhead. But as the degree of sharing increases, the storage overheads of JIH, PI and ASR become large and hence unfavorable when compared with NI.
- MI, NI and PI only support reverse traversal, but JI, JIH and ASR support traversals in multiple directions.
- All of the above indexing methods index on classes along a single path, hence for more
complex object retrieval that requires results from classes on multiple paths, extra processing is required.

In contrast, the aim of our SJIH framework is to support multiple path queries, facilitate traversal in both forward and reverse direction, and at the same time be extensible for accommodating new user requirements.
Chapter 3

Query Driven Vertical Class Partitioning

Object Oriented Database (OODB) technology is being used to support mainstream business information systems and decision support systems. In both kinds of systems it is critical not only to have efficient implementation but also a good design of the database. Vertical partitioning is a design technique for reducing the number of disk accesses needed for executing a query by minimizing the number of irrelevant instance variables accessed. This is accomplished by grouping the frequently accessed instance variables as vertical class fragments. The complexity of object oriented database models due to the class inheritance hierarchy and the class composition hierarchy complicates the definition and representation of vertical partitioning of the classes, and makes the problem of vertical partitioning in OODBs very challenging. In contrast to vertical partitioning, indexing in OODBs is a facility to reduce the number of disk IOs in query processing (e.g., see [7,51,55,56]), by reducing the accesses to irrelevant object instances. Note that while indexing reduces disk IOs at the object instance level, it may still access some irrelevant instance variables (i.e., not all the instance variables accessed will always be relevant to the query). While vertical partitioning is also a facility in OODBs to reduce the number of disk IOs in query processing, it does so by reducing the accesses to irrelevant fragments containing irrelevant instance variables. Hence, partitioning reduces disk IOs at the instance variable level. In a nutshell, indexing is complementary to vertical partitioning, and indices can be specified on vertical fragments to further reduce irrelevant object instance retrievals. In this chapter and [30,31], we develop a comprehensive analytical cost model for processing of queries on vertically class partitioned OODB schema. Analytical experimental results are presented to show the effectiveness of vertical partitioning, and the trade-offs between projection ratio vs. selectivity factor vis-a-vis sequential vs. indexed access.

This chapter is organized as follows: Section 3.1 presents the background to vertical class
partitioning. Section 3.2 presents the cost model for query processing and its evaluation. Section 3.3 presents results of analytical experiments conducted to show the utility and benefit of vertical partitioning. Section 3.4 presents the cost model for update. Section 3.5 presents comparison between affinity-based and cost-based approaches. Section 3.6 presents analysis on query execution cost before and after VCP. Finally, Section 3.7 presents our summary of this chapter.

3.1 Background

We start with reviewing the background concepts on representing vertical fragments internally (and transparently), which is necessary for our subsequent discussion. In this thesis and in [22, 18, 49], we concentrate on non-overlapping vertical partitioning, where there is no overlap in the attributes of the vertical fragments.

3.1.1 Internal Representation of Vertical Fragments

Basically there are two approaches in representing vertical fragments:

- **Object-based Class Approach**: A composite object $\overline{C}$ that contains the reference pointers to the $k$ vertical fragments $(V_1, V_2, \ldots, V_k)$ is created as shown in Figure 11. All the instance variables of class $\overline{C}$ are OIDs.

- **Embedded Class Approach**: In this case, the less frequently accessed instance variables are stored as separate vertical fragments and are linked to the composite object $\overline{D}$ by means of OIDs. The other very frequently accessed instance variables $iv1$ to $ivh$ are embedded inside the composite object $\overline{D}$, which means composite object $\overline{D}$ will have embedded instance variables and linked vertical fragments. Thus, $\overline{D}$ is an embedded class with both object-based and (very frequently accessed) value-based instance variables.

The object-based class approach uses the technique of linking from the composite object to its vertical fragments while the embedded class approach uses both linking and embedding techniques. From the design point of view, the object-based class approach is simpler than the embedded class approach, due to its clear semantics between class $\overline{C}$ and class $C$. Also from the storage point of view the object length of class $\overline{C}$ is fixed for a given vertical partitioning scheme, and an access to an instance variable of class $C$ always requires two class accesses. This homogeneity in accesses simplifies the cost models and facilitates management and maintenance of the vertical class fragments. This is especially true when the vertical class partitioning scheme can change. Further, it allows us to defer the decision of which representation to use until after the vertical class
fragments are generated. In this thesis, we shall therefore concentrate on the object-based class approach as it is more general than the embedded class approach.

Figure 11: Object-based Class Approach of internal representation of vertical class fragments

Figure 12: Embedded Class Approach of internal representation of vertical class fragments

3.2 Query Processing Cost model

In this section, we present a general analytical cost model for processing a query over both unpartitioned classes and vertically partitioned classes. We further highlight the differences between the
cost models in these two cases and discuss the overhead due to vertical partitioning. This cost model will be used for analytical experiments in section 3.3.

3.2.1 Cost model components

The total cost of processing a query is given by:

\[ \text{Total\_cost} = \text{IO\_cost} + \text{CPU\_cost} \]

where \( \text{IO\_cost} \) is the cost for performing disk IOs and \( \text{CPU\_cost} \) is the cost for performing computation during query processing. In this thesis, as in [22], we concentrate on the \( \text{IO\_cost} \) and disregard the \( \text{CPU\_cost} \). This is because for very large database applications with a huge amount of data accesses, the \( \text{CPU\_cost} \)'s contribution towards the \( \text{Total\_cost} \) will not be significant.

We assume the syntax of Object Query Language (an Object Oriented version of SQL, e.g., see [35,36]) as given below:

```
SELECT "result list"
FROM "target class"
WHERE "condition/predicate"
```

The "result list" can involve instance variables from any class in the schema. The "condition" can involve predicates on instance variables from any class. In addition to that, the condition may also allow us to specify a path expression along the class composition hierarchy. A path expression is of the form \( c_1 \cdot a_2 \cdot a_3 \cdot a_4 \cdot \ldots \cdot a_n \cdot vbiv \), where \( c_1 \) is the root class, \( a_i \) is the object based instance variable defined in class \( c_{i-1} \) with domain of class \( c_i \) with \( 2 \leq i \leq n \). An implicit join condition is of the form \( c_1 \cdot a_2 \cdot a_3 \cdot a_4 \cdot \ldots \cdot a_n \cdot vbiv \ relop \ const \), where \( relop \) can be any relational operator: ";=", "\<\", ";>\", ";<\", ";\geq\", and ";\leq\", and \( const \) is a constant value in the domain of the value based instance variable \( vbiv \) of class \( c_n \). A predicate with a path expression is executed by recursively processing the class composition hierarchy corresponding to the path using depth first search (as in [37]).

We make the following distinctions regarding the amount of main memory available for query processing since main memory availability affects the number of disk accesses:

- **Large Memory Hypothesis (LMH):** the main memory size is so large that we have enough memory buffers for all the incoming objects (i.e., in loading objects from the disk, they are only loaded once) and
- **Small Memory Hypothesis (SMH):** the main memory size is so small that we can afford to allocate only one page of memory buffer for each class or fragmented class (i.e., during the predicate evaluation, the same objects or object fragments of a particular class may be required to be loaded into the main memory multiple times and cause an high increase in the number of disk IOs).
Our cost model is based on a set of parameters which can be categorized into three types as shown in Table 2, namely, database parameters, query parameters, and specific vertical partitioning parameters.

<table>
<thead>
<tr>
<th>Category</th>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>|C_{i,k}|</td>
<td>cardinality of class collection (C_{i,k}) (i.e., (k) th subclass of (i) th class along the class composition hierarchy)</td>
</tr>
<tr>
<td></td>
<td>|C_{i,k}|</td>
<td>number of pages occupied by class (C_{i,k})</td>
</tr>
<tr>
<td></td>
<td>SC_{i,k}</td>
<td>size of object (in bytes) of class (C_{i,k})</td>
</tr>
<tr>
<td></td>
<td>(q_i)</td>
<td>number of subclasses in the class inheritance hierarchy rooted by class (C_i)</td>
</tr>
<tr>
<td></td>
<td>fan_{i-1,i,k}</td>
<td>fan-out for the class composition hierarchy from (j) th subclass of class (C_{i-1}) to the (k) th subclass of class (C_i)</td>
</tr>
<tr>
<td></td>
<td>(n)</td>
<td>path length of the path expression, i.e., the number of classes along the path expression in the class composition hierarchy</td>
</tr>
<tr>
<td></td>
<td>NP_{i,k}</td>
<td>number of objects (of (k) th subclass of class (C_i)) per page. If (SC_{i,k} &lt; PS) then (NP_{i,k} = \left\lfloor \frac{PS}{SC_{i,k}} \right\rfloor), otherwise we set it to 1</td>
</tr>
<tr>
<td></td>
<td>(b)</td>
<td>B*-tree index average fan-out</td>
</tr>
<tr>
<td></td>
<td>PS</td>
<td>page size of the file system (in unit of byte)</td>
</tr>
<tr>
<td>Query</td>
<td>(M_{i,j,k})</td>
<td>a binary variable, 1 if the query accesses (j) th vertical fragment of the (k) th subclass in class (C_i); 0 otherwise</td>
</tr>
<tr>
<td></td>
<td>Sproj_{i,k}</td>
<td>length of output result that is within (k) th subclass of class (C_i)</td>
</tr>
<tr>
<td></td>
<td>Sproj_{i,j,k}</td>
<td>length of output result that is within (j) th fragment of the (k) th subclass of class (C_i)</td>
</tr>
<tr>
<td></td>
<td>ref_{i,k}</td>
<td>number of object references for (k) th subclass in class (C_i) during the path expression evaluation process along the class composition hierarchy</td>
</tr>
<tr>
<td></td>
<td>SEL_{i}</td>
<td>selectivity of the query's predicate on class (C_i)</td>
</tr>
<tr>
<td>Vertical Partitioning</td>
<td>(m_i)</td>
<td>number of fragments in class (C_i)</td>
</tr>
<tr>
<td></td>
<td>SC_{i,k}V_j</td>
<td>size of (j) th fragment in (k) th subclass in class (C_i)</td>
</tr>
<tr>
<td></td>
<td>|C_{i,k}V_j|</td>
<td>number of pages occupied by (j) th fragment of the (k) th subclass of class (C_i)</td>
</tr>
<tr>
<td></td>
<td>VP_i</td>
<td>a binary variable, it is of value 1 if class/subclasses of (C_i) are vertically partitioned; 0 otherwise</td>
</tr>
<tr>
<td></td>
<td>|CO_{i,k}|</td>
<td>number of pages occupied by the composite objects of class (C_{i,k})</td>
</tr>
<tr>
<td></td>
<td>SCO_{i,k}</td>
<td>size of composite object (in unit of byte) in class (C_{i,k})</td>
</tr>
</tbody>
</table>

| Running Variables | \(k\) | for classes/subclasses within a class inheritance hierarchy rooted by a class \(C_i\), from 0 to \(q_i\) |
|                   | \(j\) | for the fragments of a class, from 1 to \(m_i\) |
|                   | \(i, r\) | for all classes in the class composition hierarchy/path expressions, from 1 to \(n\) |
Basic cost factors

(a) Estimation of the number of pages in a class collection

The total number of pages occupied by a class collection $C$ with object size $SC$ and cardinality $|C|$ is given by: $|C| = \lceil \frac{|C| \times SC}{PS} \rceil$, where $\lceil \cdot \rceil$ is the ceiling function and $PS$ is the page size used by the OODB system. When applying these formulae to a class inheritance hierarchy, we assume that objects of the same class/subclass are stored together, but among different classes/subclasses, objects are stored separately. This means all the subclasses are not clustered into one huge class collection for the reason of efficient processing of queries on individual subclasses. When we apply these formulae to a class inheritance hierarchy with vertical fragments, we also assume that the fragments of different class/subclasses are stored separately (for the same reason as the unpartitioned case). The same assumptions are also used for storing composite objects.

(b) Estimation of the number of page accesses required to select a certain number of records

The Yao function [78] is used to estimate the number of page accesses to select a certain number of records. Given $n$ records uniformly distributed into $b$ blocks or pages ($1 < b \leq n$), each containing $n/b$ records. If $k$ records ($k \leq n$) are randomly selected from the $n$ records, the expected number of page accesses is given by $Yao(n, b, k) = b \left( 1 - \prod_{i=1}^{k} \frac{n - i + 1}{n - i + 1} \right)$, where $d = 1 - \frac{1}{b}$. The expected number of page accesses is not equal to $k$ because some pages may contain two or more result records. In our discussion, we take $n = |C|$, $b = |C|$ and $k = SEL \times |C|$ (where $SEL$ is the selectivity of the predicate on the current class $C$). But the Yao function is only applicable when $b \leq n$, that is when object size is smaller than or equal to the page size. For object size greater than page size, we estimate the number of page accesses by a simple proportion: $b/k/n$. In building the cost model, we therefore use the auxiliary function $Y$:

$$Y(n, b, k)=\begin{cases} Yao(n, b, k) & \text{for object size smaller or equal to page size} \\ b/k/n & \text{for object size larger than page size} \end{cases}$$

(c) Estimation of the number of page accesses for index lookup

If the predicate in the query involves an instance variable associated with an index, we can make use of this index to expedite the loading of root class objects. For clustered index B*-tree with average fan-out $b$ [21,37], the number of page accesses required is $lo_{b_k} \left( SEL_1 \times \sum_{k=0}^{q_1} \frac{|C_{1,k}|}{NP_{1,k}} \right)$ to lookup a clustered index, where $SEL_1$ is the selectivity of the predicate on the root class and $NP_{1,k}$ is the num-
ber of objects (of the \( k \) th subclass of the root class) per page. For non-clustered index [21,37], the
number of page accesses required is \( \log_b \left( SEL_1 \times \sum_{k=0}^{q_1} ||C_{1,k}|| \right) \).

(d) Estimation of the number of object references
We need to estimate the number of object references during predicate evaluation (along the class composition hierarchy). For sequential scan, \( ref_{i,k} = ||C_{1,k}|| \) for \( 0 \leq k \leq q_1 \) and
\[
ref_{i,k} = \left( \sum_{j=0}^{q_{i-1}} ref_{i-1,j} \times fan_{i-1,i,j,k} \right) \times SEL_{i-1} \quad \text{for} \quad 1 < i \leq n \quad \text{and} \quad 0 \leq k \leq q_i.
\]
For clustered index scan,
\[
ref_{i,k} = SEL_1 \times ||C_{1,k}|| \quad \text{for} \quad 0 \leq k \leq q_1, \quad ref_{2,k} = \sum_{j=0}^{q_1} ref_{1,j} \times fan_{1,2,j,k} \quad \text{for} \quad 0 \leq k \leq q_2, \quad \text{and}
\]
\[
ref_{i,k} = \left( \sum_{j=0}^{q_{i-1}} ref_{i-1,j} \times fan_{i-1,i,j,k} \right) \times SEL_{i-1} \quad \text{for} \quad 2 < i \leq n.
\]
For non-clustered index scan, the \( ref_{i,k} \)'s are the same as that for clustered index scan.

3.2.2 Unpartitioned (UP) classes
Along with a class hierarchy for the unpartitioned case, there is a class composition hierarchy along the path from class \( C_1, C_2 \), through \( C_n \). And for every class \( C_i \), there is also a class inheritance hierarchy rooted by it. Our cost model is quite general in that these different class inheritance hierarchies may have a different number of child nodes; further, they can also have a different number of tree levels in the hierarchy. We denote the \( k \)th subclass of the class inheritance hierarchy (rooted by class \( C_i \)) by the notation \( C_{i,k} \), where \( k \) ranges over 1 through \( q_i \) (the total number of subclasses in class \( C_i \)). To make the cost formulae more compact, we denote the root class \( C_i \) as \( C_{i,0} \). The cost model can be broken up into 3 components: the cost of loading a class collection, the cost of evaluating the predicate, and the cost of building the output result. We note that the unpartitioned class cost model is similar to [37]'s formulation.

The total IO cost of query processing consists of three components:
\[
Total_{IO}_{UP} = IO_{Load} + IO_{Eval} + IO_{Build}
\]
where \( IO_{Load} \) is the number of page accesses to load the root class objects to start the path expression traversal, \( IO_{Eval} \) is the number of page accesses to traverse the path expression along the class composition hierarchy, and \( IO_{Build} \) is the number of page accesses to generate the result. To start the path expression traversal, we need to first load in the root class objects. There are two scan strategies: (a) sequential scan, and (b) index scan. These two strategies have different cost formulae for
the IOLoad component. But the scan strategy to load root class objects does not affect the path traversal in the other classes nor the subsequent building of results. Hence, the IOEval and IOBuild cost formulae are the same no matter which scan strategy is used. The cost formulae are presented in Tables 3 and 4, and the detailed derivation is described below.

The following is the detailed derivations of the cost model for unpartitioned classes:

**LMH case**

**IOLoad**

(a) Sequential scan

The whole root class collection is sequentially scanned. The number of page accesses required to load the root class objects using sequential scan is given by:

\[
IOLoad(\text{Seq}) = \sum_{k=0}^{q_1} |C_{1,k}| \text{ where } q_1 \text{ is the number of subclasses in the class hierarchy rooted by } C_{1,0}.
\]

(b) Index scan

If the predicate in the query involves an instance variable associated with an index, we can make use of this index to expedite the loading of root class objects. For clustered index with average B+-tree fan-out \(b \) [21,76,37]:

\[
IOLoad(\text{CluIndex}) = \log_b \left( \frac{SEL_1 \times \sum_{k=0}^{q_1} |C_{1,k}|}{\sum_{k=0}^{q_1} NP_{1,k}} \right) + \sum_{k=0}^{q_1} Y(\|C_{1,k}\|, |C_{1,k}|, SEL_1 \times \|C_{1,k}\|)
\]

where \(SEL_1\) is the selectivity of the predicate on the root class. The term \(\log_b \left( \frac{SEL_1 \times \sum_{k=0}^{q_1} |C_{1,k}|}{\sum_{k=0}^{q_1} NP_{1,k}} \right)\) represents the number of page accesses to look up the clustered index. And the term \(\sum_{k=0}^{q_1} Y(\|C_{1,k}\|, |C_{1,k}|, SEL_1 \times \|C_{1,k}\|)\) represents the number of page accesses to load in the objects derived from the index look up. It is a summation over all subclasses. Further it involves the \(Y\) function (extended Yao function) as one target page may contain more than one relevant object. For non-clustered index [21,76,37]:

\[
IOLoad(\text{NonCluIndex}) = \log_b \left( \frac{SEL_1 \times \sum_{k=0}^{q_1} |C_{1,k}|}{\sum_{k=0}^{q_1} PS_{C_{1,k}}} \right) + SEL_1 \times \sum_{k=0}^{q_1} \left( \|C_{1,k}\| \times \left\lfloor \frac{SC_{1,k}}{PS} \right\rfloor \right)
\]
The term \( \log_{2} \left( \text{SEL}_1 \times \sum_{k=0}^{q_1} \| C_{1,k} \| \right) \) represents the number of page accesses to look up the non-clustered index. And the term \( \text{SEL}_1 \times \sum_{k=0}^{q_1} \left( \| C_{1,k} \| \times \frac{SC_{1,k}}{PS} \right) \) represents the number of page accesses to load in the objects derived from the index look up. It is a summation over all subclasses, but unlike the clustered index case, it does not involve the \( Y \) function. This is because in a non-clustered index, the data objects are not ordered in the data pages, hence the \( Y \) function cannot be applied.

**IOEval**

In estimating the number of page accesses to a collection during the evaluation of a predicate, we use the \( Y \) function. The number of page accesses in predicate evaluation during the traversal along the whole path expression is given by:

\[
\text{IOEval} = \sum_{i=2}^{n} \left( \sum_{k=0}^{q_i} \text{Y}(\| C_{i,k} \|, |C_{i,k}|, \text{ref}_{i,k}) \right)
\]

The outer summation is over the classes along the class composition hierarchy, which ranges from 2 to \( n \). It starts from 2, not 1, as the scanning of the root class collection is already incorporated in the term \( \text{IOLoad} \). The inner summation is over the different subclasses in the IsA class hierarchy rooted by class \( C_{i,0} \). The term \( \text{ref}_{i,k} \) is the number of object references in class \( C_{i,k} \) during the predicate evaluation.

**IOBuild**

The number of page accesses in building the result is given by:

\[
\text{IOBuild} = \sum_{i=1}^{n} \left( \sum_{k=0}^{q_i} \text{SEL}_1 \times \| C_{i,k} \| \times \frac{S\text{proj}_{i,k}}{PS} \right)
\]

Similar to \( \text{IOEval} \), the outer summation is over the class along the class composition hierarchy, but this time, it ranges from 1 to \( n \). The inner summation is still over the different subclasses in the IsA class hierarchy rooted by class \( C_{i,0} \). The result in page accesses from class \( C_{i,k} \) is given by \( \text{SEL}_1 \times \| C_{i,k} \| \times \text{Sproj}_{i,k} \) divided by the page size \( PS \).

**SMH case**

For the SMH case, the \( \text{IOLoad}(\text{Seq}), \text{IOLoad}(\text{NonCluIndex}) \) and \( \text{IOBuild} \) are identical to that of the LMH case. As there is only one page in buffer for every class, the performance of clustered index scan deteriorates to that of non-clustered index scan:

\[
\text{IOLoad}_{\text{SMH}}(\text{CluIndex}) = \text{IOLoad}_{\text{SMH}}(\text{NonCluIndex}) = \text{IOLoad}_{\text{LMH}}(\text{NonCluIndex})
\]

32
Further, the \textit{IOEval} is quite different from the LMH case:

\[
\text{IOEval}_{\text{SMH}} = \sum_{i=2}^{n} \left[ \sum_{k=0}^{q_1} \left( \prod_{r=2}^{i} \text{ref}_{r,k} \times \left[ \frac{SC_{i,k}}{PS} \right] \right) \right]
\]

The cost of evaluating the predicate is the sum of the page accesses for the different class collections along the path expression. In the \textit{i}th class collection along the path expression, we need

\[
\sum_{k=0}^{q_1} \left( \prod_{r=2}^{i} \text{ref}_{r,k} \times \left[ \frac{SC_{i,k}}{PS} \right] \right)
\]

page accesses. Thus, SMH requires many more pages to be loaded into main memory than LMH.

\textbf{Table 3: Component-wise cost model formulae for LMH case}

<table>
<thead>
<tr>
<th>LMH</th>
<th>Unpartitioned</th>
<th>Vertically Partitioned</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{IOLoad} (Seg)</td>
<td>$\sum_{k=0}^{q_1}</td>
<td>C_{1,k}</td>
</tr>
<tr>
<td>\textit{IOLoad} (CluIndex)</td>
<td>log$<em>b \left( SEL_1 \times \sum</em>{k=0}^{q_1}</td>
<td>C_{1,k}</td>
</tr>
<tr>
<td>\textit{IOLoad} (NonCluIndex)</td>
<td>log$<em>b \left( SEL_1 \times \sum</em>{k=0}^{q_1}</td>
<td>C_{1,k}</td>
</tr>
<tr>
<td>\textit{IOEval}</td>
<td>$\sum_{i=2}^{n} \sum_{k=0}^{q_1} Y(C_{i,k},</td>
<td>C_{i,k}</td>
</tr>
<tr>
<td>\textit{IOBuild}</td>
<td>$\frac{\sum_{i=1}^{n} \sum_{k=0}^{q_1} \left[ SEL_1 \times</td>
<td>C_{i,k}</td>
</tr>
</tbody>
</table>
Table 4: Component-wise cost model formulae for SMH case

<table>
<thead>
<tr>
<th>SMH</th>
<th>Unpartitioned</th>
<th>Vertically Partitioned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1OLoad (Seq)</td>
<td>same as LMH</td>
<td>same as LMH</td>
</tr>
<tr>
<td>1OLoad (CluIndex)</td>
<td>[ \log_2 \left( \frac{\text{SEL}<em>1 \times \sum</em>{k=0}^{q_1} |C_{1,k}|}{PS} \right) + \log_2 \left( \frac{\text{SEL}<em>1 \times \sum</em>{k=0}^{q_1} |C_{1,k}|}{PS} \right) + \frac{\text{SEL}<em>1 \times \sum</em>{k=0}^{q_1} |C_{1,k}|}{PS} ]</td>
<td>[ \log_2 \left( \frac{\text{SEL}<em>1 \times \sum</em>{k=0}^{q_1} |C_{1,k}|}{PS} \right) + \frac{\text{SEL}<em>1 \times \sum</em>{k=0}^{q_1} |C|}{PS} ]</td>
</tr>
<tr>
<td>1OLoad (NonCluIndex)</td>
<td>same as LMH</td>
<td>same as LMH</td>
</tr>
<tr>
<td>1OEval</td>
<td>[ \sum_{i=2}^{n} \sum_{k=0}^{q_i} \left[ \left( \frac{\text{SC}_{1,k}}{PS} \right)^i \right] ]</td>
<td>[ \sum_{i=2}^{n} \sum_{k=0}^{q_i} \left[ \left( \frac{\text{SC}_{1,k} \times \text{V}}{PS} \right)^i \right] ]</td>
</tr>
<tr>
<td>1OBuild</td>
<td>same as LMH</td>
<td>same as LMH</td>
</tr>
</tbody>
</table>

### 3.2.3 Vertically partitioned (VP) classes

We adopt an object oriented representation of vertical partitioning: each of the vertical class fragments is represented as a class, and a logical object of class \( C \) is internally represented as a composite object (of class \( \overline{C} \)) which contains pointers to the vertical class fragments.

Figure 13 shows a vertically partitioned class hierarchy. Note that not all classes along the path expression need to be vertically partitioned. We define binary variables \( VP_i \), with

\[
VP_i = \begin{cases} 
1 & \text{if class } C_i \text{ is vertically partitioned} \\
0 & \text{otherwise}
\end{cases}
\]

where \( 1 \leq i \leq n \). We denote the \( j \)-th vertical fragment of the \( k \)-th subclass of class \( C_i \) by \( C_{i,k}V_j \), where \( j \) ranges over 1 through \( m_i \) (the total number of fragments for each class in the class inheritance hierarchy rooted by class \( C_i \)). We assume after vertical partitioning, that all the class/subclasses of the whole class inheritance hierarchy are partitioned into the same number of fragments for simplicity\(^1\).

Given a predicate/condition in a query, we define binary variables \( M_{i,j,k} \), where

---

1. A subclass can have more or fewer instance variables than its superclass, hence a subclass can have more or fewer number of fragments. Our cost model can easily be extended to include such cases, but this will make the cost formulae more complicated.
\[ M_{i,j,k} = \begin{cases} 
1 & \text{if the query accesses the } j \text{th vertical fragment of} \\
\text{the } k \text{th subclass in class } C_i \\
0 & \text{otherwise} 
\end{cases} \]

for \( 1 \leq i \leq n, \ 1 \leq j \leq m_i \) and \( 0 \leq k \leq q_i \).

The cost formulae are presented in Tables 3 and 4, and the detailed derivations are described below. Note that our cost model is a general-purpose one which supports the main OODB features, including (a) **Class composition hierarchy**: Class composition hierarchy is fully supported in our cost model in the form of path expression traversal; (b) **Class inheritance hierarchy**: The summation of cost over class inheritance hierarchy is incorporated, so superclass/subclass hierarchy is fully supported in our cost model. The hierarchy can be of any level and subclasses can define new instance variables and may have different object sizes.

![Diagram](image.png)

Figure 13: Schema for a path expression in vertically partitioned case

The following is the detailed derivation of the cost model for vertically partitioned classes:

**LMH case**

**IOLoad**

(a) Sequential scan

For loading the root class collection, we need to load in all the pages of the root class composite object collection. At the same time, we also need to load in the pages of the vertical fragments. But we need only to load in those vertical fragments that are relevant to the query (i.e., with \( M_{i,j,k} = 1 \)). As there are \( m_1 \) vertical fragments in the root class, the \( IOLoad(Seq) \) contains a component which
is a summation over all the vertical fragments.

\[
IOLoad(\text{Seq}) = \sum_{k=0}^{q_1} \left[ |CO_{1,k}| + \sum_{j=1}^{m_1} M_{1,j,k} \times |C_{1,k}V_j| \right]
\]

It is a summation over all the subclasses. The term \( |CO_{1,k}| \) represents the extra-overhead to handle the composite objects generated from VCP. The term \( \sum_{j=1}^{m_1} M_{1,j,k} \times |C_{1,k}V_j| \) represents the number of page accesses to load in the relevant vertical fragments and it involves a summation over all the \( m_1 \) vertical fragments.

(b) Index scan

\[
IOLoad(\text{CluIndex}) = \log_b \left( SEL_1 \times \sum_{k=0}^{q_1} \frac{||C_{1,k}||}{NP_{1,k}} \right) + \sum_{k=0}^{q_1} \left( \sum_{j=1}^{m_1} M_{1,j,k} \times Y(||C_{1,k}||, |C_{1,k}V_j|, \text{ref}_{1,k}) \right)
\]

The term \( \log_b \left( SEL_1 \times \sum_{k=0}^{q_1} \frac{||C_{1,k}||}{NP_{1,k}} \right) \) represents the number of page accesses to look up the clustered index. The term \( \sum_{k=0}^{q_1} Y(||C_{1,k}||, |C_{1,k}V_j|, \text{ref}_{1,k}) \) represents the extra-overhead to handle the composite objects generated from vertical partitioning and it involves a \( Y \) function as the composite objects are stored together and sorted in OID sequence. The term \( \sum_{k=0}^{q_1} \left( \sum_{j=1}^{m_1} M_{1,j,k} \times Y(||C_{1,k}||, |C_{1,k}V_j|, \text{ref}_{1,k}) \right) \) represents the number of page accesses to load in the objects derived from the index look up. It is a summation over all subclasses and all vertical fragments. Further, it involves the \( Y \) function as one target page may contain more than one relevant object.

\[
IOLoad(\text{NonCluIndex}) = \log_b \left( SEL_1 \times \sum_{k=0}^{q_1} ||C_{1,k}|| \right) + SEL_1 \times \sum_{k=0}^{q_1} |CO_{1,k}| +
\]

\[
SEL_1 \times \left[ \sum_{k=0}^{q_1} \sum_{j=1}^{m_1} M_{1,j,k} \times |C_{1,k}V_j| \times \left[ \frac{SC_{1,k}V_j}{PS} \right] \right]
\]

The term \( \log_b \left( SEL_1 \times \sum_{k=0}^{q_1} ||C_{1,k}|| \right) \) represents the number of page accesses to look up the non-clustered
index. The term $SEL_i \times \sum_{k=0}^{q_i} |CO_{i,k}|$ represents the extra-overhead to handle the composite objects generated from vertical partitioning. The term $SEL_i \times \left[ \sum_{k=0}^{q_i} \sum_{j=1}^{m_{i,j,k}} \frac{SC_{i,k}V_j}{PS} \right]$ represents the number of page accesses to load in the objects derived from the index look up. It is a summation over all subclasses and all vertical fragments.

**IOEval**

Similar to the unpartitioned case, the *IOEval* is a sum of *Y* functions over the class composition hierarchy, but now we need to consider the effect of vertical partitioning of the classes along the path expression (which is indicated by the binary variables $VP_i$):

$$IOEval = \sum_{i=2}^{n} \left( \sum_{k=0}^{q_i} \left[ VP_i \times \left[ Y(\|C_{i,k}\|, |CO_{i,k}|, ref_{i,k}) + \sum_{j=1}^{m_{i,j,k}} Y(\|C_{i,k}\|, |C_{i,k}V_j|, ref_{i,k}) \right] \right] + (1 - VP_i) \times Y(\|C_{i,k}\|, |C_{i,k}|, ref_{i,k}) \right)$$

The outer summation is over all classes along the class composition hierarchy and the inner summation is over all subclasses. If the class hierarchy is vertically partitioned ($VP_i = 1$), the term $Y(\|C_{i,k}\|, |CO_{i,k}|, ref_{i,k})$ represents the extra overhead incurred from processing the composite objects and the term $\sum_{j=1}^{m_{i,j,k}} (M_{i,j,k} \times Y(\|C_{i,k}\|, |C_{i,k}V_j|, ref_{i,k}))$ represents the number of page accesses to load in the relevant vertical fragments. If the class hierarchy is not vertically partitioned ($VP_i = 0$), the term $Y(\|C_{i,k}\|, |C_{i,k}|, ref_{i,k})$ is similar to the unpartitioned case and represents the number of page accesses to load in all the unpartitioned objects.

**IOBuild**

The *IOBuild* is also a summation over the fragments that give the result fragment pages:

$$IOBuild = \sum_{i=1}^{n} \sum_{k=0}^{q_i} \left[ \sum_{j=1}^{m_{i,j,k}} \frac{SEL_i \times |C_{i,k}| \times Spr_{i,j,k}}{PS} \right]$$

It is a summation over all classes along the class composition hierarchy, all subclasses in the IsA class hierarchy and all vertical fragments.

**SMH case**

For the SMH case, the *IOLoad(Seq), IOLoad(NonCluIndex)* and *IOBuild* are identical to that of the LMH case. As there is only one page in buffer for every class, similar to the unpartitioned case, the performance of clustered index scan deteriorates to that of non-clustered index scan:
IOLoad_SMH(\text{CluIndex}) = IOLoad_SMH(\text{NonCluIndex}) = IOLoad_LMH(\text{NonCluIndex})

= \log_2 \left( \sum_{k=0}^{q_1} \left[ SEL_1 \times \sum_{k=0}^{q_1} C_{1,k} \right] + SEL_1 \times \sum_{k=0}^{q_1} C_{1,k} \right) + \sum_{k=0}^{q_1} M_{1,j,k} \times \left[ C_{1,k} \times \frac{SC_{1,k} V_j}{PS} \right]

Further, the \text{IOEval} is quite different from LMH case:

\text{IOEval}_\text{SMH} = \sum_{i=2}^{n} \left( \prod_{r=2}^{i} ref_{r,k} \right) \times \left( 1 + \sum_{j=1}^{m_i} M_{i,j,k} \times \frac{SC_{i,k} V_j}{PS} \right)

It is a summation over all classes along the class composition hierarchy and all subclasses in the \text{IsA} class hierarchy. The cost of evaluating the predicate is a sum of page accesses for the different class collections along the path expression. In the \text{i}th class collection along the path expression, we need \sum_{k=0}^{q_1} \left( \prod_{r=2}^{i} ref_{r,k} \right) \times \left( 1 + \sum_{j=1}^{m_i} M_{i,j,k} \times \frac{SC_{i,k} V_j}{PS} \right) page accesses. The "1" in the term

1 + \sum_{j=1}^{m_i} M_{i,j,k} \times \frac{SC_{i,k} V_j}{PS}

represents the extra overhead to process the composite objects generated from vertical partitioning, as there is not enough buffer space, every object/fragment access requires one such page access to the composite object. Thus, SMH requires many more pages to be loaded into main memory than LMH.

3.2.4 Guidelines for beneficial vertical partitioning schemes

The following is a detailed discussion on the overheads and guidelines for vertical partitioning. In this section, we compare the cost formulae for the unpartitioned case with the vertically partitioned case to evaluate the overhead and utility of vertical partitioning. Then, we suggest some guidelines for the use of vertical partitioning. We shall compare these cost formulae component by component and discuss the overhead caused by vertical partitioning. We concentrate on discussing the LMH case first.

LMH case

IOLoad(Seq)

The term \sum_{k=0}^{q_1} |C_{0,1,k}| in the vertical partitioning (VP) case is caused by the loading of the composite objects generated from vertical partitioning. It is an extra overhead when compared with the unpartitioned (UP) case. For large objects, this overhead is relatively small when compared with the loading of the large-sized vertical fragments. But in the case of small objects and the highly fragmented case (that is the object is fragmented into a large number of small fragments), this overhead will
blow up and cause the VP case to have poorer performance than the UP case.

\[ |C_{1,k}| = \sum_{j=1}^{m_1} |C_{1,k}V_j|, \text{ i.e., the number of pages occupied by the unpartitioned class collection} \]

\( C_{1,k} \) is approximately equal to the sum of the number of pages occupied by the vertical fragments \( C_{1,k}V_j \) (for \( 1 \leq j \leq m_1 \) and \( 0 \leq k \leq q_1 \)). This implies that:

(1) If only a few fragments are relevant to a query, that is, \( \sum_{j=1}^{m_1} M_{1,j,k} \) is close to 1, then

\[ \sum_{j=1}^{m_1} M_{1,j,k} \times |C_{1,k}V_j| \] will be just the sum of pages occupied by a few vertical fragments and will be much smaller than \( |C_{1,k}| \). Therefore, VP is beneficial;

(2) On the other hand, if the query requires most of the fragments, that is, \( \sum_{j=1}^{m_1} M_{1,j,k} \) is close to \( m_1 \), then

\[ \sum_{j=1}^{m_1} M_{1,j,k} \times |C_{1,k}V_j| \] will be very close to \( |C_{1,k}| \). Therefore, VP is not so beneficial;

(3) A good vertical partitioning scheme should minimize \( \sum_{j=1}^{m_1} M_{1,j,k} \times |C_{1,k}V_j| \). This can be done by either minimizing \( |C_{1,k}V_j| \), that is, the number of pages occupied by the relevant vertical fragments (as the storage space of a vertical fragment is proportional to the size of the fragmented object, so the size of the fragmented object should be as small as possible), or, by minimizing \( \sum_{j=1}^{m_1} M_{1,j,k} \), that is, we should try to group the relevant instance variables into as few vertical fragments as possible.

**IOLoad(ClueIndex)**

Similar to the sequential scan case, the term \( \sum_{k=0}^{q_1} Y([C_{1,k}], [CO_{1,k}], ref_{1,k}) \) in the VP case represents the extra overhead to process the composite objects generated from VP. This overhead is relatively unimportant for large objects but will become more and more important as the size of object decreases and/or the number of vertical fragments increases. Both UP and VP involve the term

\[ \log_b \left( SEL_{1} \times \sum_{k=0}^{q_1} \frac{|C_{1,k}|}{MP_{1,k}} \right) \] which represents the number of page accesses to look up the clustered index.
As \( Y(||C_{i,k}||, |C_{i,k}|, ref_{i,k}) = \sum_{j=1}^{m_i} Y(||C_{i,k}||, |C_{i,k}V_j|, ref_{i,k}) \), the arguments given above in \( IOLoad(Seq) \) also hold.

**IOLoad(NonC1uIndex)**

Similar to the sequential scan case, the term \( \sum_{k=0}^{q_1} |CO_{i,k}| \) in the VP case represents the extra overhead to process the composite objects generated from VP. Both UP and VP involve the term \( \log_b \left( SEL_1 \times \sum_{k=0}^{q_1} ||C_{i,k}|| \right) \) which represents the number of page accesses to look up the non-clustered index. As \( ||C_{i,k}|| \times \left[ \frac{SC_{i,k}}{PS} \right] = \sum_{j=1}^{m_i} ||C_{i,k}|| \times \left[ \frac{SC_{i,k}V_j}{PS} \right] \) the arguments in \( IOLoad(Seq) \) also hold.

**IOEval**

The term \( \sum_{i=2}^{n} \left( \sum_{k=0}^{q_i} (Y(||C_{i,k}||, |CO_{i,k}|, ref_{i,k}) \right) \) in the VP case represents the extra overhead to process the composite objects generated from VP. For a particular \( i \) along the class composition hierarchy, if \( VP_i = 0 \) then the cost of \( IOEval \) will be the same for both the vertically partitioned case and unpartitioned case in that class. The more interesting case is with \( VP_i = 1 \), then the \( IOEval \) of the VP case will become

\[
\sum_{k=0}^{q_i} \left[ Y(||C_{i,k}||, |CO_{i,k}|, ref_{i,k}) + \sum_{j=1}^{m_i} M_{i,k} \times Y(||C_{i,k}||, |C_{i,k}V_j|, ref_{i,k}) \right].
\]

As \( Y(||C_{i,k}||, |C_{i,k}|, ref_{i,k}) = \sum_{j=1}^{m_i} Y(||C_{i,k}||, |C_{i,k}V_j|, ref_{i,k}) \) the arguments in \( IOLoad(Seq) \) also hold.

Further, if the fan-outs between the classes in the class composition hierarchy are high, the term \( IOEval \) will dominate the total IO cost, then it is advisable to use vertical partitioning along the whole path expression.

**IOBuild**

As \( \sum_{i=1}^{n} \left[ \sum_{k=0}^{q_i} \sum_{j=1}^{m_i} Sproj_{i,k} \right] = \sum_{i=1}^{n} \left[ \sum_{k=0}^{q_i} \sum_{j=1}^{m_i} Sproj_{i,k} \right] \), that means during the building of results, VP will not improve the performance of query execution.
SMH case

As the process of sequential scanning the root class collection requires only one page buffer, therefore, the availability of main memory will not affect the term $IOLoad(\text{Seq})$. With similar argument, $IOLoad(\text{NonCluIndex})$ and $IOBuild$ will also be not affected.

$IOLoad(\text{CluIndex})$

In SMH case, there is only one page buffer for every class/class fragment, the performance of clustered index scan deteriorates to that of non-clustered index scan. The comparison between the UP and VP case is similar to that of $IOLoad(\text{NonCluIndex})$ in LMH case.

$IOEval$

The availability of main memory has a great effect on the term $IOEval$. The term "1" in the VP case represents the extra overhead to process the composite objects generated from VP. As

$$SC_{i,k} = \sum_{j=1}^{m_i} SC_{i,k}V_j$$

the arguments in $IOLoad(\text{Seq})$ also hold.

Guidelines for beneficial vertical partitioning schemes

A beneficial vertical partitioning scheme should:

- minimize the number of pages occupied by the relevant vertical fragments (also the size of the vertical class fragment object should be as small as possible);
- try to group the relevant instance variables into as few vertical class fragments as possible;
- use vertical partitioning along the complete path expression, if the fan-outs between the classes in the class composition hierarchy is high.

3.3 Analytical experiments

To compare and contrast the utility of vertical partitioning, we conducted in this section a number of analytical experiments to evaluate the effect of number of vertical fragments, projection ratio, fan-out, cardinality and selectivity on the performance gain due to vertical partitioning. In today’s technology, main memory size of a computer system is increasing. Furthermore, SMH’s assumption of only one buffer page per class/fragment represents a overly pessimistic estimation. So in the following discussion, we shall concentrate on LMH.

3.3.1 Performance metric

The improvement of performance is characterized by the performance metric, $Normalized IO$ (NIO) where:

$$Normalized IO = \frac{\text{Number of Disk IO for Vertical Partitioned class(es) case}}{\text{Number of Disk IO for Unpartitioned class(es) case}}$$
A beneficial vertical partitioning scheme is a scheme with NIO as less than 1. To study the impact of parameter changes while maintaining the control over the number of parameters, we consider the following seven parameters. These are cardinality of root class, page size, number of objects per page, number of vertical fragments per class, fan-out of a class along class composition hierarchy, selectivity of predicate, and projection ratio (discussed later) of a query. Further, in these experiments, we consider a class collection with a class composition hierarchy of path length 3. That is, there are three classes $C_1$, $C_2$, and $C_3$, with a class composition hierarchy $C_1 \rightarrow C_2 \rightarrow C_3$.

3.3.2 Experiment parameters

The parameter settings for each of the experiment are as follows:

- cardinalities: for a particular class $C_i$, we assume that all the subclasses of $C_i$ have the same cardinality as $C_i$, but a different class $C_j$ can have different cardinality;

- size of objects: for each experiment, objects in all the class/subclasses are of the same size;

- size of fragments: for each experiment, we assume that the size of fragments is the same (i.e., we divide each class into equal length fragments) for all class/subclasses;

- number of fragments: for each experiment, the number of fragments is the same for all classes/subclasses;

- number of subclasses in an IsA class hierarchy: we take the number of subclasses ($q_i$) rooted by class $C_i$ to be 3 for $1 \leq i \leq 3$, i.e., there are all together 4 classes/subclasses including the root class;

- fan-outs: $fan_{i-1,i,j,k} = 0$ for $j \neq k$ and $2 \leq i \leq n$, and all $fan_{i-1,i,k,k}$ are the same for $2 \leq i \leq n$ and $0 \leq k \leq q_i$. From this, if we define $\|C_{i,\text{Hier}}\| = \sum_{k=0}^{q_i} \|C_{i,k}\|$, the total cardinality of the whole IsA class hierarchy and let $fan_{i-1,i,k,k} = \text{FAN}$, then $\|C_{2,\text{Hier}}\| = \|C_{1,\text{Hier}}\| \times \text{FAN}$ and $\|C_{3,\text{Hier}}\| = \|C_{2,\text{Hier}}\| \times \text{FAN}$;

- results: we assume that the projected instance variables for query results are only from the root class $C_1$;

- selectivities: as the results are from the root class, we concentrate on the selectivity on the root class, and for the other classes, their selectivities are set to 1. We note that this corresponds to the worst case scenario and we may over-estimate the number of disk accesses required. Further research involves detailed study of the query processing strategies used during the path expression traversal. These strategies may include depth-first search, breadth-first search, top-down evaluation or bottom-up evaluation. Detailed study on query processing strategies are outside the scope of this thesis.
• projection ratio: the projection ratio is defined as the ratio between the length of the relevant instance variables (on a projection of instance variables of a class) and the length of the original objects. In the cost model, we need to obtain the values for the sum of some cost components over all fragments, e.g., the sum \( \sum_{j=1}^{m_i} M_{i,j,k} \) in the expression \( \sum_{j=1}^{m_i} M_{i,j,k} \times |C_{i,k} V_j| \). Without any query characteristics/information, we make the following simplification in calculating the sum: if \( m \) is the number of fragments, the sum should be a value between 1 and \( m \). A sum of value 1 is not always possible. This is because for large projection ratio (\( PR \)), one fragment cannot contain all the result instance variables. We use the formula \( \text{sum} = \lceil m \times PR \rceil \). If \( m \times PR \) is small (say 0.4) we use the ceiling function to make the sum to be at least 1. On the other hand, if \( PR \) is large (say 0.99) we have the other extreme that the sum is just \( m \).

• The average \( B^+ \)-tree fan-out is set as 100.

3.3.3 Effect of varying the number of fragments

In this section, we investigate the utility of vertical partitioning by identifying the existence of an optimal number of fragments that can give rise to high performance gain. We perform two analytical experiments: the first one concentrates on sequential scan, and the second considers both sequential and index scans. These two experiments are based on a very general cost model that treats all attributes alike (that is, they have the same size and so on). Hence, we do not need any query details except two important query characteristics: the selectivity and the projection ratio.

Sequential scan

In this experiment, we concentrate on the sequential scan strategy to the root class collection. We want to identify the existence of the optimal number of fragments that produces high performance gain. In this experiment, we consider three possible schemes for vertical partitioning:

• \( VP1 \) - only class \( c_1 \) is vertically partitioned;

• \( VP2 \) - both classes \( c_1 \) and \( c_2 \) are vertically partitioned;

• \( VP3 \) - all the classes \( c_1, c_2 \) and \( c_3 \) are vertically partitioned.

The reason for selecting such a class collection is that it enables us to study both the impact of fan-out along the class composition hierarchy, and also the effect of partitioning classes along the class composition hierarchy. The parameter values used and experimental result plots are shown in Figures 14 and 15, and in particular, the number of objects per page is 1/16 (for Figure 14) and 16 (for Figure 15). The size of the object varies from 0.5KByte to 128KBytes. The number of fragments for class \( c_1 \) are 2, 4, 8 and so on.
In Figures 14 and 15, the legends $VPI_i$ for $1 \leq i \leq 3$ are as given above, and $S$ is the selectivity of the predicates of the query.

After conducting the experiments, we have the following observations from Figures 14 and 15:

- All curves are parabolic in shape and show a minimum Normalized IO at a certain optimal number of fragments. In Figure 14, the performance gain for different numbers of fragments ranges from the best: 76.2% (0.762=1-0.238, for $VP_3$ with high (0.95) selectivity) savings, which is quite substantial; to the worst: -162.2% (-1.622=1-2.622, for $VP_3$ with low (0.05) selectivity) where it needs 2.622 times the total IO cost of the unpartitioned case. The above results imply that the optimal number of fragments exists and a good choice of the number of fragments can produce an optimal partitioning scheme with high performance gain. The plots for other numbers of objects per page, fan-outs and projection ratios show similar results.

- The plots of Normalized IO vs. Number of Fragments show that the Normalized IO decreases initially as the number of fragments initially increase, but starts to increase as the number of fragments further increase, revealing an optimal number of vertical fragments. This can be explained by the fact that the vertical fragments are stored independently, i.e., different class fragments are stored in different disk pages. A large number of vertical fragments leads to small fragments, and accesses to a large number of small fragments cause extra disk IOs. Further, if there is a large number of fragments, the total storage size of composite objects will also be large and hence the overhead to process these composite objects will also be large. Therefore, the performance gain will decrease (due to overhead) and the Normalized IO value will start to increase. In some cases, the Normalized IO value for a very large number of fragments can jump above 2 as shown in Figure 14, resulting in a very poor performance for vertical partitioning in comparison to the unpartitioned case.

- We observe that for any particular selectivity, if the classes are not excessively vertically partitioned into too many very small vertical fragments, the NIO values show the trend:

$$NIO \ VP_1 > NIO \ VP_2 > NIO \ VP_3$$

implying that $VP_3$ is the best in terms of performance gain. The observation is that one should vertically partition all the classes along the class composition hierarchy, especially for high fan-out cases.

**Sequential and index scan**

In the previous experiment, we concentrated on sequential scan strategy. In this experiment, we want to show that the optimal number of fragments also exists for the other scan strategies (namely,
clustered index scan and non-clustered index scan) and we want to compare the performance gains due to these scan strategies.

We repeat the previous experiment but include the clustered and non-clustered index scans. In the following experiments, we only consider VP3 (that is, with all the classes along the class composition hierarchy being vertically partitioned) to concentrate on the comparison of different scan strategies to the root class collection. Parameter values used to produce experimental results (as shown in Figures 16 and 17) are similar to the previous experiment with the object size being 0.5KBytes (for Figure 16) and 128KBytes (for Figure 17), respectively. The legends in Figures 16 and 17 mean the following: SEQ means sequential scan, CI means clustered index scan, NCI means non-clustered index scan for the root class, and S is the selectivity and with the same meaning as that of Figures 14 and 15.

![Graph showing normalized IO vs. number of fragments](image)

**Figure 14:** NIO vs. Number of Fragments (for number of object per page = 1/16, fan-out = 2.0, projection ratio = 0.2)
Based on Figures 16 and 17, we have the following observations:

- No matter which scan strategy we use, the curves in Figures 16 and 17 show that different scan strategies always have an optimal number of fragments which can give rise to substantial performance gain for vertical partitioning.

- When comparing the different scan strategies, sequential scan is more flexible and has a larger range of number of fragments that can produce good performance gain. While for the clustered and non-clustered index scan, they have a smaller range of number of fragments that can produce good performance gain. Further, both are more sensitive to the increase in the number of fragments; at higher number of fragments, the NIO values easily go above 1, implying poorer performance than the unpartitioned case.

- When comparing Figures 16 and 17, we notice that for large objects (128KB), we can vertically fragment it to a larger number of fragments before the NIO value exceeds 1. Further, a large object is not as sensitive to the variation in the number of fragments as a small object.
Figure 16: NIO vs. Number of Fragments for different scan strategies (for number of object per page = 1/16, fan-out = 2.0, projection ratio = 0.2)

Figure 17: NIO vs. Number of Fragments for different scan strategies (for number of object per page = 16, fan-out = 2.0, projection ratio = 0.2)
3.3.4 Effect of varying the fan-outs

In this experiment, we study the impact of the variations in the fan-outs (along class composition hierarchy) on the performance gain. The main purpose of this experiment is to show the performance gain for vertical partitioning for high fan-out values. That is, great savings can be obtained by using vertical partitioning at high fan-out. In Figure 18, the number of fragments in class $c_1$ is 32 and in the curve of VP3, the normalized IOs approach $1/32 = 0.03$. That is, a performance gain of 97% saving, which is quite substantial in query execution. Note that with high fan-out, the number of object instances in classes $c_2$ and $c_3$, i.e., $|C_2|$ and $|C_3|$ will be quite large when compared to $|C_1|$. Therefore, the IOEval will dominate the total IO cost. For the VP3 case (with low projection ratio), all classes in $c_1$, $c_2$ and $c_3$ are vertically partitioned into 32 fragments. During the predicate evaluation, we need only to retrieve 1 out of the 32 fragments from classes $c_1$, $c_2$ and $c_3$ (as we only need the fragment that contains the relevant instance variables). Note also that the curves for VP1 and VP2 approach an NIO value of 1.0 at high fan-outs. The observation is that if the fan-outs between the classes in the class composition hierarchy are high, it is advisable to use vertical partitioning along the whole path expression.

3.3.5 Effect of varying the cardinalities

In this experiment, we study the impact of the variations of the cardinality of the root class on the performance gain. The main result of this experiment is that the performance gain in terms of normalized IO is constant as the cardinality increases. All the curves in Figure 19 are almost horizontal straight lines, showing that the normalized IO is independent of the cardinality of the root class. That is, irrespective of the cardinality of the root class we get the same percentage of reduction in number of disk accesses.
Cardinality = 1000
Page Size = 8192
Number Of Class/Subclasses = 4
Number Of Object Per page = 0.062500
Number of Fragments = 32
Projection Ratio = 0.010

<table>
<thead>
<tr>
<th>Fan-out</th>
<th>VP1</th>
<th>VP2</th>
<th>VP3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>0.438</td>
<td>0.172</td>
<td>0.039</td>
</tr>
<tr>
<td>1</td>
<td>0.668</td>
<td>0.352</td>
<td>0.037</td>
</tr>
<tr>
<td>2</td>
<td>0.856</td>
<td>0.582</td>
<td>0.036</td>
</tr>
<tr>
<td>4</td>
<td>0.962</td>
<td>0.768</td>
<td>0.035</td>
</tr>
<tr>
<td>8</td>
<td>0.986</td>
<td>0.880</td>
<td>0.035</td>
</tr>
<tr>
<td>16</td>
<td>0.996</td>
<td>0.940</td>
<td>0.034</td>
</tr>
<tr>
<td>32</td>
<td>0.999</td>
<td>0.970</td>
<td>0.034</td>
</tr>
</tbody>
</table>

Figure 18: NIO vs. Fan-outs

Page Size = 8192
Number Of Class/Subclasses = 4
Number Of Object Per page = 0.062500
Number of Fragments = 8
Projection Ratio = 0.200
Fan-out = 2.00
Cardinality | VP1   | VP2   | VP3   |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.428</td>
<td>0.371</td>
<td>0.258</td>
</tr>
<tr>
<td>10000</td>
<td>0.428</td>
<td>0.371</td>
<td>0.258</td>
</tr>
<tr>
<td>100000</td>
<td>0.428</td>
<td>0.371</td>
<td>0.258</td>
</tr>
<tr>
<td>1000000</td>
<td>0.428</td>
<td>0.371</td>
<td>0.258</td>
</tr>
</tbody>
</table>

Plot of Normalized IO vs. Cardinalities

Figure 19: NIO vs. Cardinalities
3.3.6 Trade off between projection ratio and selectivity

We now turn our attention to one of the most influential factors on the utility of vertical partitioning, that is, the projection ratio. We show that projection ratio determines whether vertical partitioning is beneficial or not. We further compare and contrast the results from pure vertical partitioning and pure indexing. By pure vertical partitioning, we mean that we use sequential scan on the root class collection (but the classes $C_1$, $C_2$ and $C_3$ are vertically partitioned) and by pure indexing, we just use a clustered index scan on the root class collection (but all classes $C_1$, $C_2$ and $C_3$ are not vertically partitioned). By performing these experiments, we obtain more insight on the utility of vertical partitioning as opposed to indexing. Note that as we are only considering a clustered index scan on the root class collection, this experiment has the limitation of not considering other ways of indexing the three classes.

Effect of varying the projection ratio

In this experiment we study the impact of the variation of projection ratio on the NIO which indicates the performance gain. In the plots in Figures 20 and 21, the parameter values are as shown in the figures. Both of the figures have the same parameter settings except for the object size (128KB and 0.5KB for Figures 20 and 21, respectively), both of the figures show the following trend:

- For low projection ratio, vertical partitioning produces high performance gain.
- As projection ratio increases, the performance gain diminishes.
- In Figure 20, at PR=0.9, NIO reaches 1.0. In Figure 21, at PR near 0.8, NIO reaches 1.0.

Effect of varying the projection ratio and selectivity

From the previous experiment, we observe that projection ratio is an important factor which governs the usability of vertical partitioning. In this experiment, we further study the impact of the variation of projection ratio so as to compare pure vertical partitioning and pure indexing. At each fixed projection ratio, we determine the selectivity at which pure vertical partitioning NIO will be the same as pure indexing NIO, and for selectivity higher than that, pure vertical partitioning NIO will be lower than pure indexing NIO.
Figure 20: NIO vs. Projection Ratio (for object size 128KB)

Figure 21: NIO vs. Projection Ratio (for object size 0.5KB)
In the 3D plots in Figures 22 to 25, the parameter values are presented on top of the plots. In particular, the fan-out is 2.0 (for Figures 22 and 23) and 0.125 (for Figures 24 and 25), the number of fragments is 8 (for Figures 22 and 24) and 32 (for Figures 23 and 25). The projection ratio varies from 0.01, 0.1, 0.2, ..., 0.9 to 0.99. The object size varies from 0.5KB, 8KB to 128KB. The observations from Figures 22 to 25 are:

- The region above the surface in the 3D plot is the region in which pure vertical partitioning performs better than pure indexing. Hence the larger area in the upper region implies vertical partitioning is more widely applicable (hence more effective) than indexing. That is, the parameter ranges for which vertical partitioning out performs indexing is much larger than the parameter range over which indexing outperforms vertical partitioning.

- From all the four figures, there is a large region in which vertical partitioning performs better than indexing. The conclusion is therefore that vertical partitioning is a complementary technique for saving disk IOs.

- When comparing Figure 22 with Figure 24, the only difference is the fan-out. In Figure 22 the fan-out is 2.0 and it shows a larger upper region than that of Figure 24 (with fan-out 0.125), implying that Figure 22 has better performance gain in vertical partitioning than Figure 24. The observation is that high fan-out favours vertical partitioning. Figures 23 and 25 show similar results.

![3D plot of Projection Ratio vs. Object Size vs. Selectivity (for number of fragments = 8, fan-out = 2.0)](image)

Figure 22: 3D plot of Projection Ratio vs. Object Size vs. Selectivity (for number of fragments = 8, fan-out = 2.0)
Figure 23: 3D plot of Projection Ratio vs. Object Size vs. Selectivity (for number of fragments = 32, fan-out = 2.0)

Figure 24: 3D plot of Projection Ratio vs. Object Size vs. Selectivity (for number of fragments = 8, fan-out = 0.125)
3.3.7 Summary of results from experiments

The experiments that we conducted present us with the following results: (1) There is an optimal number of vertical fragments for a class collection, and vertical partitioning can give rise to substantial savings in the number of disk accesses; (2) The projection ratio is one of the most influential factors that determines whether vertical partitioning is beneficial or not; (3) The fan-out parameter has an impact on the vertical partition algorithms. If there are queries that access objects using path expressions, then it is preferable to vertically fragment all the classes in the class composition hierarchy. This results in considerable IO savings for low fan-outs with the IO savings increasing with the increase in the fan-out; (4) The cardinality of the class does not impact the normalized IO savings, implying that as the cardinality of the classes increases the IO savings will also proportionally increase; and (5) Vertical partitioning is a good technique for saving disk accesses when compared with indexing.

Based on the above results we foresee a need for cost-based algorithms for vertical partitioning of a class collection. In the next section, we present two contrasting approaches to vertical partitioning.
3.4 Cost Model for Update

When compared with updates on unpartitioned classes, updates on vertically partitioned classes will require different number of disk accesses. In this section, we will present the update cost model for the three update operations: (a) insert, (b) delete, and (c) update. We discuss the update operation towards a group of objects that satisfy the predicate in an OQL statement.

3.4.1 Unpartitioned classes

Insert

To insert a group of objects into an unpartitioned OODB, the number of disk accesses required to write the objects onto the disk is equal to the number of disk accesses required to read in the objects from the disk during query processing. The IO cost for the insert consists of two components:

\[ \text{Insert}_{\text{IO UP}} = \text{IOWrite} + \text{IOEval} \]

where \( \text{IOWrite} \) is the number of disk accesses to write the root class objects. \( \text{IOEval} \) is the number of disk accesses to write the objects and traverse the path expression along the class composition hierarchy. The term \( \text{IOWrite} \) is equal to the \( \text{IOLoad} \), and the term \( \text{IOEval} \) is equal to that in query processing on unpartitioned classes (as discussed in Section 3.2.2).

Delete

To perform physical deletion on a group of objects from an unpartitioned OODB, the number of disk accesses required to read in the deleted objects from the disk is equal to the number of disk accesses required to read in the objects during query processing. The IO cost for the delete consists of also two components:

\[ \text{Delete}_{\text{IO UP}} = \text{IORead} + \text{IOEval} \]

where \( \text{IORead} \) is the number of disk accesses to read in the root class target objects for deletion. \( \text{IOEval} \) is the number of disk accesses to read in the target objects for deletion and traverse the path expression along the class composition hierarchy. The term \( \text{IORead} \) is equal to the \( \text{IOLoad} \), and the term \( \text{IOEval} \) is equal to that in query processing on unpartitioned classes (as discussed in Section 3.2.2).

Update

To update a group of objects in an unpartitioned OODB, there are two possibilities:

- (a) update on value-based instance variable (VBIV)
- (b) update on object-based instance variable (OBIV)

To simply the presentation, we present cost formulae for update of the root class objects when only one instance variable is updated. For update of other classes along the class composition hierarchy and/or more than one instance variables, the cost formulae can similarly be derived.

(a) Update on one value-based instance variable (VBIV)
To update an VBIV of a group of root class objects in an unpartitioned OODB, we need to read in the root class target objects and then write the updated objects back to the disk (as only root class objects are updated). The IO cost for the update consists of two components:

\[ \text{Update}_{\text{IO\_UP}} = \text{IRead} + \text{IWrite} \]

where \(\text{IRead}\) is the number of disk accesses to read in the root class target objects. \(\text{IWrite}\) is the number of disk accesses to write the updated objects back to disk. The terms \(\text{IRead}\) and \(\text{IWrite}\) are both equal to the \(\text{IOLoad}\) in query processing on unpartitioned classes (as discussed in Section 3.2.2).

(b) Update on one object-based instance variable (OIV)

To update an OIV of a group of root class objects in an unpartitioned OODB, we further have the following two sub-cases: (i) the updated OIV will point to an existing object, and (ii) the updated OIV will point to a newly created object.

Case (i)

In this case, as the updated OIV will point to an existing object, we need only to read in the root class target objects and then write back the updated root class objects. The IO cost for the update consists of two components:

\[ \text{Update}_{\text{IO\_UP}} = \text{IRead} + \text{IWrite} \]

Similar to (a) for the VBIV, the terms \(\text{IRead}\) and \(\text{IWrite}\) are both equal to the \(\text{IOLoad}\) in query processing on unpartitioned classes (as discussed in Section 3.2.2).

Case (ii)

In this case, the updated OIV of the root class object will point to a newly created object. To simply the presentation, we present an update that will cause objects to be created in all the classes along the class composition hierarchy (except the root class). The cost formulae for updates that will cause objects to be created in only some of the classes can similarly be derived. In this case, we not only read the root class target objects and then write the updated root class objects, the update also incurs disk accesses to write the newly created objects. Therefore, the IO cost for the update consists of three components:

\[ \text{Update}_{\text{IO\_UP}} = \text{IRead} + \text{IWrite} + \text{IEval} \]

The terms \(\text{IRead}\) and \(\text{IWrite}\) are both equal to the \(\text{IOLoad}\) in query processing on unpartitioned classes (as discussed in Section 3.2.2). \(\text{IEval}\) is the number of disk accesses to write the newly created objects and traverse the path expression along the class composition hierarchy. The term \(\text{IEval}\) is equal to that for query processing on unpartitioned classes (as discussed in Section 3.2.2). If the fan-outs along the class composition hierarchy is high, \(\text{IEval}\) will increase rapidly, causing a high update cost.
3.4.2 Vertically partitioned classes

With vertically partitioned classes, we need to take care of the update operations towards the composite objects and vertical class fragments created after vertical partitioning.

**Insert**

To insert a group of objects into a vertically partitioned OODB, the number of disk accesses required to write the objects onto the disk is equal to the number of disk accesses required to read in the objects from the disk. The IO cost for the insert consists of two components:

\[ \text{Insert}_{\text{IO}_{\text{VP}}} = \text{IOWrite} + \text{IOEval} \]

where \( \text{IOWrite} \) is the number of disk accesses to write the root class composite objects and vertical class fragments. \( \text{IOEval} \) is the number of disk accesses to write the composite objects and vertical class fragments and traverse the path expression along the class composition hierarchy. The term \( \text{IOWrite} \) is equal to the \( \text{IOLoad} \), and the term \( \text{IOEval} \) is equal to that in query processing on vertically partitioned classes (as discussed in Section 3.2.3).

**Delete**

To delete a group of objects from a vertically partitioned OODB, the number of disk accesses required to read in the deleted composite objects and vertical fragments from the disk is equal to the number of disk accesses required to read in the composite objects and vertical fragments during query processing. The IO cost for the delete consists of two components:

\[ \text{Delete}_{\text{IO}_{\text{VP}}} = \text{IOracle} + \text{IOEval} \]

where \( \text{IOracle} \) is the number of disk accesses to read in the root class target composite objects and vertical fragments for deletion. \( \text{IOEval} \) is the number of disk accesses to read in the target composite objects and vertical fragments for deletion and traverse the path expression along the class composition hierarchy. The term \( \text{IOracle} \) is equal to the \( \text{IOLoad} \), and the term \( \text{IOEval} \) is equal to that in query processing on vertically partitioned classes (as discussed in Section 3.2.3).

**Update**

Similar to the unpartitioned classes case, to update a group of objects in a vertically partitioned OODB, there are two possibilities:

- (a) update on value-based instance variable (VBIV)
- (b) update on object-based instance variable (OBIV)

**(a) Update on one value-based instance variable (VBIV)**

To update an VBIV of a group of root class objects in a vertically partitioned OODB, we need to read in the root class target composite objects and the target vertical class fragment for the update and then write the updated root class vertical class fragment back to the disk (as only root class objects are updated and we only need the vertical class fragment that contains the VBIV being updated). The IO cost for the update consists of two components:
Update \(_{IO_{VP}}=IOR\_read+IOWr\_ite\)

where \(IOR\_read\) is the number of disk accesses to read in the root class target composite objects. \(IOWr\_ite\) is the number of disk accesses to write the updated objects back to disk.

We first present the cost formulae for LMH case. Depending on the scan strategies, we have the following three cost formulae for \(IOR\_read\). For sequential scan, \(IOR\_read\) is

\[
\sum_{k=0}^{q_1} \left( |C_{1,k}| + |C_{1,k} \cdot V_u| \right),
\]

with the vertical class fragment \(C_{1,k} \cdot V_u\) containing the VBIU being updated. For clustered index scan, \(IOR\_read\) is

\[
\log_b \left( \text{SEL}_1 \times \sum_{k=0}^{q_1} |C_{1,k}| \right) + \sum_{k=0}^{q_1} Y( |C_{1,k}|, |C_{1,k} \cdot V_u|, ref_{1,k} ) + \sum_{k=0}^{q_1} Y( |C_{1,k}|, |C_{1,k} \cdot V_u|, ref_{1,k} ),
\]

where \(\text{SEL}_1\) is the selectivity of the predicate for the update OQL statement, \(ref_{1,k}\) is the number of references on class \(C_{1,k}\) (and is derived in Section 3.2.1). For non-clustered index scan, \(IOR\_read\) is

\[
\log_b \left( \text{SEL}_1 \times \sum_{k=0}^{q_1} |C_{1,k}| \right) + \text{SEL}_1 \times \sum_{k=0}^{q_1} |C_{1,k}| + \text{SEL}_1 \times \left[ \sum_{k=0}^{q_1} |C_{1,k}| \times \left( \frac{SC_{1,k} \cdot V_u}{PS} \right) \right],
\]

where \(SC_{1,k} \cdot V_u\) is the size of the vertical class fragment that contains the VBIU being updated. Depending on the scan strategies, we also have the following three cost formulae for \(IOWr\_ite\). For sequential scan, \(IOWr\_ite\) is

\[
\sum_{k=0}^{q_1} |C_{1,k} \cdot V_u|.
\]

For clustered index scan, \(IOWr\_ite\) is

\[
\sum_{k=0}^{q_1} Y( |C_{1,k}|, |C_{1,k} \cdot V_u|, ref_{1,k} ).
\]

For non-clustered index scan, \(IOWr\_ite\) is

\[
\text{SEL}_1 \times \left[ \sum_{k=0}^{q_1} |C_{1,k}| \times \left( \frac{SC_{1,k} \cdot V_u}{PS} \right) \right].
\]

We then present the cost formulae for the SMH case. For sequential scan, \(IOR\_read\) is the same as LMH case and is

\[
\sum_{k=0}^{q_1} \left( |C_{1,k}| + |C_{1,k} \cdot V_u| \right).
\]

For both clustered and non-clustered index scan, \(IOR\_read\) are

\[
\log_b \left( \text{SEL}_1 \times \sum_{k=0}^{q_1} |C_{1,k}| \right) + \text{SEL}_1 \times \sum_{k=0}^{q_1} |C_{1,k}| + \text{SEL}_1 \times \left[ \sum_{k=0}^{q_1} |C_{1,k}| \times \left( \frac{SC_{1,k} \cdot V_u}{PS} \right) \right].
\]

For sequential scan, \(IOWr\_ite\) is the same as LMH case and is

\[
\sum_{k=0}^{q_1} |C_{1,k} \cdot V_u|.
\]

For both clustered and non-clustered index scan, \(IOWr\_ite\) are

\[
\text{SEL}_1 \times \left[ \sum_{k=0}^{q_1} |C_{1,k}| \times \left( \frac{SC_{1,k} \cdot V_u}{PS} \right) \right].
\]

(b) Update on one object-based instance variable (OBIV)

To update an OBIV of a group of root class objects in a vertically partitioned OODB, similar to the
unpartitioned classes case, we have the following two sub-cases: (i) the updated OBIV will point to an existing composite object, and (ii) the updated OBIV will point to a newly created composite object.

Case (i)
In this case, as the updated OBIV will point to an existing composite object, we need only to read in the root class target composite objects and vertical fragments and then write back the updated root class vertical fragments. Similar to (a) above (for update on VBIV), the IO cost for the update consists of two components:

\[ \text{Update}_{\text{IO}}\text{-VP} = \text{IORead} + \text{IOWrite} \]

The terms \text{IORead} and \text{IOWrite} are both equal to that of (a) discussed above.

Case (ii)
In this case, the updated OBIV of the root class object will point to a newly created object. We present the update that will cause composite objects and vertical class fragments to be created in all the classes along the class composition hierarchy (except the root class). In this case, we are not only required to read in the root class target composite objects and vertical class fragments and then write back the updated root class vertical class fragments, we further require disk accesses to write the newly created composite objects and vertical class fragments onto the disk. The IO cost for the update consists of three components:

\[ \text{Update}_{\text{IO}}\text{-VP} = \text{IORead} + \text{IOWrite} + \text{IOEval} \]

The terms \text{IORead} and \text{IOWrite} are both equal to that of Case (i). \text{IOEval} is the number of disk accesses to write the newly created composite objects and vertical class fragments and traverse the path expression along the class composition hierarchy. For LMH case, \text{IOEval} is

\[
\sum_{i=2}^{n} \sum_{k=0}^{q_i} \{ V_P \times Y(\|C_{i,k}\|, \|CO_{i,k}\|, \text{ref}_{i,k}) + \sum_{j=1}^{m_i} Y(\|C_{i,k}\|, \|C_{i,k}V_j\|, \text{ref}_{i,k}) + (1 - V_P) \times Y(\|C_{i,k}\|, \|C_{i,k}\|, \text{ref}_{i,k}) \}.
\]

For SMH case, \text{IOEval} is

\[
\sum_{i=2}^{n} \sum_{k=0}^{q_i} \left[ \prod_{\tau=2}^{r} \text{ref}_{i,k} \right] \times \left( 1 + \sum_{j=1}^{m_i} \frac{SC_{i,k}V_j}{PS} \right) \].
\]

Similar to the unpartitioned case, if the fan-outs along the class composition hierarchy is high, \text{IOEval} will increase rapidly, causing a high update cost.

3.5 Vertical Class Partitioning Approaches for Query Execution
There are two main approaches/techniques for vertical partitioning, namely, affinity-based [18,62] and cost-based [22]. In [18], it was shown that the optimal partitioning scheme generated by different affinity-based partitioning techniques are sometimes different even for the same input. Furthermore, an affinity-based cost model was developed to evaluate the performance of different affinity-
based partitioning techniques. It was shown that their Minimum Square Error (MSE) performed the best. Though this approach was developed for relational database systems, it is equally applicable for object oriented databases. This MSE approach will be compared against the cost-based approach developed in this thesis. Note that we are evaluating two techniques for vertical partitioning, and we do this by exhaustively enumerating all the possible vertical fragments and finding the best vertical partition generated by both the approaches. The MSE partitioning technique can at best equal the number of disk accesses given by the optimal partitioning scheme generated by the cost-based technique.

3.5.1 Minimum Square Error (MSE) Vertical Partitioning Technique

The input is an attribute usage matrix which consists of the attribute in a relation as columns and the queries as rows with the frequency of access to the attribute for each query as the values in the matrix. In this model, there are two terms to measure the goodness of the vertical partitioning scheme, one is the irrelevant attribute access cost and the other one is the relevant attribute access cost. The irrelevant attribute access cost is represented by the square error between the actual access patterns of the query in the partitioning scheme and the mean vector (which represents the average access pattern of the queries over all attributes of the fragments). The relevant access cost computes a penalty factor that measures the contribution due to access of relevant attributes that are dispersed in different fragments. As in [18], we shall use the following notation:

- $A_{i,j}$ is the attribute vector for attribute $j$ in fragment $i$.
- $V_i$ is the mean vector for the $i$th fragment. It represents an average access pattern of the queries over all attributes of fragment $i$.
- $n_i$ is the number of attributes in fragment $i$.
- $M$ is the total number of fragments of a class.
- $T$ is the vector transpose operation.
- $NTX$ is the total number of queries that are under consideration.
- $freq_t$ is the frequency of query $t$.
- $|R_{ik}|$ is the number of relevant attributes in fragment $k$ accessed with respect to fragment $i$ by query $t$.
- $n'_{ik}$ is the total number of attributes that are in fragment $k$ accessed with respect to fragment $i$ by query $t$.
- $E^2_i$ is the irrelevant attribute access cost.
- $E^2_k$ is the relevant attribute access cost.
• $E^2$ is the square error.

The cost formula for irrelevant attribute access cost is:

$$E^2_I = \sum_{i=1}^{M} \sum_{j=1}^{n} (A_{i,j} - V_i)^T (A_{i,j} - V_i)$$

where $V_i = \frac{1}{n_i} \sum_{j=1}^{n_i} A_{i,j}$

The cost formula for relevant attribute access cost:

$$E^2_R = \sum_{i=1}^{N_T} \text{Min}_{i=1}^{M} \left( \sum_{k=1}^{\text{freq}^2_k * |R_{ik}|^2 / n'_{ik}} \right)$$

The overall Square Error is given by:

$$E^2 = E^2_I + E^2_R$$

An MSE vertical partitioning procedure

With an input of attribute usage matrix, the following algorithm will be used to enumerate all possible vertical partitioning schemes to find the optimal vertical partitioning scheme for all classes in the schema:

```
Step 1: For each class in the schema
Step 2: For i = 1 to Number_of_Instance_Variables in that class
Step 3: Enumerate all i fragment(s) partitioning schemes
Step 4: Calculate the Square Error for each partitioning scheme and determine the min Square Error partitioning scheme
Step 5: EndFor
Step 6: From all of the above min Square Error i fragment partitioning schemes, determine the final overall min Square Error partitioning scheme
Step 7: EndFor
```

MSE vertical partitioning procedure

3.5.2 Cost-based Vertical Partitioning (CVP) Technique

In this approach, we present a cost model based on the number of disk accesses taken to execute a query. We evaluate the partitioning schemes based on this performance metric. In our model, we not only consider query parameters (like access frequency), but also consider database characteristics (like the length of the instance variables and fan-out) and other parameters like selectivity of the query, and the instance variables participating in the query predicates and/or the query output result list. The general cost formula with query parameters is:
\[ Total\_IO = \sum_{i=1}^{N} freq_i \times (IO\_Loading\_Cost + IO\_Eval\_Cost + IO\_Building\_Output) \]

that is, the total IO cost is a weighted sum of IO cost over all queries with weighting factor \( freq_i \).

**A CVP vertical partitioning procedure**

The basic strategy is to start with the leaf classes and vertical partition them first, then propagate the saving in irrelevant data accesses towards the root class of a class composition hierarchy.

```
Step 1. For each class (starting from the leaf classes to the root class of a class composition hierarchy)
Step 2. For i:=1 to Number_Of_Instance_Variables in that class
Step 3. Enumerate all i fragment(s) partitioning schemes
Step 4. Use our cost model to calculate the Total IO Cost for each partitioning scheme and determine the min cost partitioning scheme
Step 5. EndFor
Step 6. From all of the above min cost i fragment partitioning schemes determine the final overall min cost partitioning scheme
Step 7. EndFor
```

CVP vertical partitioning procedure

**3.5.3 Comparison of our CVP technique with MSE technique**

Both these techniques generate non-overlapping and complete vertical partitioning schemes. Unlike the CVP technique the MSE technique does not take into account class hierarchy and class composition hierarchy. The MSE technique uses the instance variable usage frequency as the sole input, but CVP technique also takes care of the database characteristics (like instance variable length, fan-out and cardinality) and other query characteristics (like path length). The MSE technique assumes that any data access is of unit cost, but our CVP technique considers detailed cost breakdown at different stages in the query execution (such as, loading of class hierarchy, predicate evaluation, and result generation). Finally to obtain the optimal partitioning scheme, both techniques use the exhaustive search strategy to enumerate all the partitioning schemes and calculate their cost to obtain the minimum. The CVP approach exploits the class composition hierarchy, in that, the non-leaf classes of the class composition hierarchy can use the partitioning of their child classes to derive better vertical class fragments. But in the MSE technique all classes are partitioned independently.

**3.5.4 Example**

We shall now take an example of a class collection with three classes (Emp, Dept, and Proj) and the class composition hierarchy of path length 3. The motivation of this experiment is to show the differences between MSE and CVP procedures in finding the optimal partitioning scheme for the same example database.
There is a class composition hierarchy starting from Emp class to Dept class and then to Proj class.

The size of Emp object is 362 (assuming an OID requires 8 bytes of storage), Dept object is 172 and Proj object is 166. The other parameters used are:

- fan-out from class Emp to Dept is 0.2 and fan-out from class Dept to Proj is 5.0
- cardinality of class Emp is 1000, cardinality of class Dept is 200, and cardinality of class Proj is 1000

### 3.5.5 Query characteristics

There are 18 queries, which are classified into 6 types, which will be used to compare both the MSE and CVP techniques for vertical partitioning. User Queries are as follows (and are illustrated in Table 5):

Type 1 (only concerning class Emp):

```sql
select EName, Qualification from Emp where EmpId = 123456;
select EmpId, EName from Emp where Qualification="MBA";
select EmpId from Emp where EName = "Alan Tam";
```

Type 2 (only concerning classes Emp & Dept):

```sql
select EName, Qualification from Emp where EmpId = 123456 and DeptInfo.DName = "Personnel";
select EmpId, EName from Emp where Qualification="MBA" and DeptInfo.DName = "Personnel";
select EmpId from Emp where EName = "Alan Tam" and DeptInfo.DName = "Personnel";
```

Type 3 (concerning all 3 classes):

```sql
select EName, Qualification from Emp where EmpId = 123456 and DeptInfo.ProjInfo.PName = "OODB";
select EmpId, EName from Emp where Qualification="MBA" and DeptInfo.ProjInfo.PName = "OODB";
select EmpId from Emp where EName = "Alan Tam" and DeptInfo.ProjInfo.PName = "OODB";
```
Type 4 (only concerning class Dept):
    select DName, DeptType from Dept where DeptId = 888;
    select DName, DAddress from Dept where DeptType<>"R&D";
    select DeptId from Dept where DeptType = "Sales";

Type 5 (only concerning classes Dept & Proj):
    select DName, DeptType from Dept where DeptId = 888 and
    ProjInfo.PName = "OODB";
    select DName, DAddress from Dept where DeptType<>"R&D" and
    ProjInfo.PName = "OODB";
    select DeptId from Dept where DeptType = "Sales" and
    ProjInfo.PName = "OODB";

Type 6 (only concerning class Proj):
    select PName, ProjType from Proj where PId = 18;
    select PName, Location from Proj where Priority>1;
    select PId, ProjType from Proj where PName = "OODB";

<table>
<thead>
<tr>
<th>Type</th>
<th>Query</th>
<th>Concerned</th>
<th>DeplInfo</th>
<th>EmpId</th>
<th>EName</th>
<th>Qualification</th>
<th>EAddress</th>
<th>freq</th>
<th>selectivity</th>
<th>path length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>RL</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>100</td>
<td>0.05</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>RL</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>50</td>
<td>0.05</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>RL</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>25</td>
<td>0.05</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Query Characteristics
<table>
<thead>
<tr>
<th>Type</th>
<th>Query #</th>
<th>Concerned with</th>
<th>ProjInfo 0</th>
<th>DeptId 1</th>
<th>DName 2</th>
<th>DeptType 3</th>
<th>DAddress 4</th>
<th>freq</th>
<th>selectivity</th>
<th>path length</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>10</td>
<td>RL</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>50</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>RL</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>5</td>
<td>0.9</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>RL</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>25</td>
<td>0.4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>RL</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>25</td>
<td>0.1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>RL</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>2</td>
<td>0.9</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>RL</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td>0.4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Query #</th>
<th>Concerned with</th>
<th>ProjType 0</th>
<th>PId 1</th>
<th>PName 2</th>
<th>Priority 3</th>
<th>Location 4</th>
<th>freq</th>
<th>selectivity</th>
<th>path length</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>16</td>
<td>RL</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>25</td>
<td>0.05</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>RL</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>25</td>
<td>0.9</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>RL</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>25</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notation**

- "path length" is the path length of the query. A path length of 1 means that the query will concentrate on the root class in the class composition hierarchy. A path length of 2 means that the query involves a path expression with length 2. A path length of 3 means that the query involves a path expression with length 3, i.e., accessing all 3 classes.
- "RL" means that the instance variable with an "X" is related with the results list.
- "P" means that the instance variable with an "X" is related with the predicate.

3.5.6 Results for Minimum Square Error (MSE) Vertical Partitioning Technique

Attribute usage matrices on classes Emp, Dept and Proj are constructed from the above query characteristics. It is the only input required by the MSE vertical partitioning procedure. The results are:

- for class Proj, it is a 2 fragment partitioning scheme: (Priority Location) (ProjType PId PName);
- for class Dept, it is a 2 fragment partitioning scheme (ProjInfo DAddress) (DeptId DName DeptType);
- and for class Emp, it is a 2 fragment partitioning scheme (DeptInfo EAddress) (EmpId
3.5.7 Results for Cost-based Vertical Partitioning (CVP) technique

When we execute our CVP vertical partitioning procedure, we have the following execution steps: the first step is to find the optimal leaf class Proj vertical partitioning scheme, while the second and the third steps are to find the optimal class Dept and root class Emp partitioning schemes. When the leaf class Proj is chosen, we enumerate all possible partitioning schemes to find the optimal partitioning scheme. In calculating the total IO cost, only transactions that access the Proj class are used. The optimal partitioning scheme is the 3 fragment partitioning scheme (PName) (ProjType PId) (Priority Location). The instance variable PName is involved in the path expression DeptInfo.ProjInfo.PName, the information that PName is in a fragment by itself (of length 50 bytes) is used in steps 2 and 3. In step 2, the class Dept is chosen. Similar to the above, when calculating the total IO cost in this part, only transactions that access the Dept class (as "root" class) are used. The optimal partitioning scheme is (DAddress) (ProjInfo DeptId) (DName DeptType). The instance variable ProjInfo is involved in the path expression DeptInfo.ProjInfo.PName, the information that ProjInfo is in a fragment with another instance variable - DeptId, with total length of 12 bytes, is used in step 3. Finally in step 3, the root class Emp is chosen. Similar to the above, when calculating the total IO cost in this part, only transactions that access the Emp class are used. The results are plotted in Figure 28. The optimal partitioning scheme is (DeptInfo) (Skill) (EAddress) (EmpId EName). As illustrated in Figure 28 the optimal partitioning scheme is a 4 fragment partitioning scheme. It is interesting to note that the values for cases with 3, 4 and 5 fragments are quite close, that means if the overhead of accessing multiple fragments is high, the partitioning scheme with 3 fragments can also be utilized (although it is not the optimal partitioning scheme) to improve performance. Figure 29 shows the scatter plot of total IO cost for the different partitioning schemes. There is a wide spread in the values for the different partitioning schemes with the same number of fragments. For example, for 4 fragment partitioning schemes, the total IO costs range from 15276 to 22268 disk accesses. From this, we can conclude that we should be careful in choosing the vertical partitioning scheme. Just limiting the number of fragments in a vertical partitioning scheme will not materialize in a performance gain.
### Min Square Error Calculation for the Emp class to find the optimal partition scheme

<table>
<thead>
<tr>
<th>Number of Queries</th>
<th>Total Number of Partition Scheme</th>
<th>Number of Fragment</th>
<th>Min Square Error</th>
<th>Min Partition Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>52</td>
<td>1</td>
<td>18566 (0 1 2 3 4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>6102 (0 4 1 2 3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>6102 (0 4 1 2 3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>17177 (0 3 4 1 2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>33700 (0 1 2 3 4)</td>
<td></td>
</tr>
</tbody>
</table>

Overall Min Square Error = 6102
Optimal Partition Scheme: (0 4 1 2 3)

![Plot of Min Square Error vs. Number of Fragments](image)

Figure 26: Min Square Error vs. Number of Fragments
Figure 27: Scatter Plot of Square Error vs. Number of Fragments
Calculation Part 3 to find the Optimal Emp class partition scheme

Cardinality = 1000  Page Size = 4096
For fanout(Emp->Dept): 0.200000  fanout(Dept->Proj): 5.000000
Number of Trans = 9  Total Number of Conf = 52
Number of Fragment  Min Total IO Cost  Min Partition Scheme
1  29821  ( 0 1 2 3 4 )
2  20225  ( 0 1 2 3 )
3  15596  ( 3 4 0 1 2 )
4  15276  ( 0 3 4 1 2 )
5  15263  ( 0 1 2 3 4 )

Overall Min Total IO Cost = 15276
Optimal Partition Scheme: ( 0 )( 3 )( 4 )( 1 2 )

Plot of Min Total IO Cost vs. Number of Fragments

Figure 28: Min Total IO Cost vs. Number of Fragments
3.5.8 Comparison of MSE technique vs. CVP technique

It is interesting to compare the values for these optimal partitioning schemes obtained from the two techniques. As shown in Table 6, the values in brackets are not the optimal values in those vertical partitioning procedures, rather they are shown here to contrast the two different techniques. From Table 6, we can draw the conclusion that the two techniques produce quite different optimal partitioning schemes for a database schema with same set of user queries and their access patterns. We do another set of experiments on class Emp, in this new set of experiments, we reduce the Skill in-
stance variable length from 200 bytes to 1 byte and keep the other parameter values unchanged. As the MSE technique does not take into the account the instance variable length, so its optimal partitioning scheme remains unchanged. But the CVP technique takes into account the instance variable length and produces a new optimal partitioning scheme: (DeptInfo) (EAddress) (EmpId EName Skill). Thus, the CVP technique adapts to the changes in the physical database system parameters and generates the optimal partitioning scheme, whereas, the MSE technique does not. Further, as shown in columns 4 and 6 of Table 6, the CVP technique finds the optimal partitioning solution which requires the least number of disk accesses. At best, the MSE technique can generate a partition which requires a minimum number of disk accesses, but this is not always guaranteed as shown in this example. On the other hand, the CVP technique always generates a partitioning scheme that minimizes the number of disk accesses.

<table>
<thead>
<tr>
<th>Class</th>
<th>MSE optimal partitioning scheme</th>
<th>square error</th>
<th>Corresponding IO cost</th>
<th>CVP optimal partitioning scheme</th>
<th>total IO cost</th>
<th>Corresponding square error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emp</td>
<td>(DeptInfo EAddress) (EmpId EName Skill)</td>
<td>6102</td>
<td>(22441)</td>
<td>(DeptInfo) (Skill) (EAddress) (EmpId EName)</td>
<td>15276</td>
<td>(17177)</td>
</tr>
<tr>
<td>Dept</td>
<td>(ProjInfo DAddress) (DeptId DName DeptType)</td>
<td>1331</td>
<td>(1415)</td>
<td>(DAddress) (ProjInfo DeptId) (DName DeptType)</td>
<td>1181</td>
<td>(4864)</td>
</tr>
<tr>
<td>Proj</td>
<td>(Priority Location) (ProjType PId PName)</td>
<td>625</td>
<td>(2775)</td>
<td>(PName) (ProjType PId) (Priority Location)</td>
<td>2725</td>
<td>(1875)</td>
</tr>
</tbody>
</table>

3.6 Analysis on Query Execution Cost Before and After VCP

In this section, we present analytical results on the query execution cost before and after VCP, and investigate what type of queries can benefit from VCP and what type of queries cannot. The database schema and query characteristics are the same as that of section 3.4.4, except, in order to avoid the weighting effect of the different frequencies for the queries, the frequencies of all queries in this experiment are set to 100. Table 7 shows the results of the experiment.
Table 7: Results on Analysis of Query Execution Cost

<table>
<thead>
<tr>
<th>Type</th>
<th>Query #</th>
<th>Path length</th>
<th>VP Cost</th>
<th>UP Cost</th>
<th>Normalized IO (VP/UP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7600</td>
<td>9500</td>
<td>0.8000</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>7900</td>
<td>9800</td>
<td>0.8061</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2300</td>
<td>9200</td>
<td>0.2500</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>7700</td>
<td>9600</td>
<td>0.8021</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>8000</td>
<td>9900</td>
<td>0.8081</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>2400</td>
<td>9300</td>
<td>0.2581</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>3</td>
<td>9000</td>
<td>10900</td>
<td>0.8257</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>3</td>
<td>9300</td>
<td>11200</td>
<td>0.8304</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>3</td>
<td>3700</td>
<td>10600</td>
<td>0.3491</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>1</td>
<td>600</td>
<td>1000</td>
<td>0.6000</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1800</td>
<td>1600</td>
<td>1.1250 *</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
<td>600</td>
<td>1000</td>
<td>0.6000</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>2</td>
<td>1900</td>
<td>2300</td>
<td>0.8261</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>2</td>
<td>3100</td>
<td>2900</td>
<td>1.0690 *</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>2</td>
<td>1900</td>
<td>2300</td>
<td>0.8261</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>1</td>
<td>2100</td>
<td>4300</td>
<td>0.4884</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>1</td>
<td>8100</td>
<td>7600</td>
<td>1.0658 *</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>1</td>
<td>2100</td>
<td>4300</td>
<td>0.4884</td>
</tr>
<tr>
<td></td>
<td>Overall</td>
<td></td>
<td>80100</td>
<td>117300</td>
<td>0.6829</td>
</tr>
</tbody>
</table>

3.6.1 Discussion
The cost ratio is defined as the disk access cost ratio between the vertical partitioned case and unpartitioned case. From the sixth column of Table 7, we observe that out of the 18 NIOs, 15 of them are lower than 1.0. These 15 NIOs are between lowest 0.25 to highest 0.83. For those three NIOs that are greater than 1.0, one of them is 13% higher than the unpartitioned case (with 200 extra disk accesses), and the other two are 7% higher than the unpartitioned case (with 200 and 500 extra disk accesses, respectively). These imply that the use of vertical class partitioning is beneficial to most of the queries. Although a few queries in the query execution environment do not benefit from vertical class partitioning, the extra overhead in processing the composite object after vertical class partitioning (200+200+500=900 disk accesses) is low when compared with the overall benefit of the whole query processing environment (the last row shows that there is a 1-0.66=34% disk access cost reduction for the whole query processing environment, i.e., a total of 37,200 disk access sav-
ing). We further note that for the cases with NIOs greater than 1.0, two of them occurred at path length equals 1 and one of them occurred at path length equals 2. We conclude that, in order to reduce disk accesses in most queries, VCP should be applied to all the classes along the path of a class composition hierarchy.

3.7 Summary

Vertical class partitioning in object oriented databases is a very challenging and relevant problem. Though this problem has been addressed in relational database systems, the complexity of an object oriented data model, with class inheritance hierarchy and class composition hierarchy, presents a need for a new approach to this problem. There are fundamentally two different approaches to vertical class partitioning, namely, affinity-based, and cost-based. In this chapter, we developed the cost model for executing a query in both unpartitioned and vertically partitioned OODB schema\(^1\).

We applied the cost model to show the utility and effectiveness of the vertical partitioning, with the conclusion that optimal vertical partitioning does mostly reduce the cost of executing a query. Further, we developed two procedures for vertical partitioning based on affinity-based approach and cost-based approach. We showed that the cost-based approach is superior to the affinity-based approach in that it guarantees the minimum number of disk accesses to process all the queries.

OODB data model provides extra modeling support for designing applications. VCP design technique reduces irrelevant data accesses and provides higher performance. Example applications that can benefit from our VCP design technique in OODB include: multimedia systems and office automation systems (with text, audio, video and compound documents), software engineering design and manufacturing engineering design (CAD/CAM) systems.

When comparing our cost-based approach and the affinity-based approach, one major disadvantage of the affinity-based approach is that it will not generate a cost effective VCP scheme. On the other hand, one limitation of our cost-based approach is that finding an optimal VCP scheme requires computation complexity of \(O(n^k)\) instead of \(O(n^2)\) for affinity-based approach. Another limitation of our cost-based approach is that it requires high quality estimation of a number of cost model parameters.

Given the parameter values of a database and a query processing environment. With the help of the cost model, we can use our VCP algorithm to set up the initial VCP scheme. When the

---

1. Members of our database group have validated the cost model by empirical experimental results through an implementation on a commercial Object Database System [75]. The results showed that the experimental query processing costs were between the lower bound (as predicted by LMH) and upper bound (as predicted by SMH) for both the unpartitioned and vertically partitioned OODB schema.
database and/or the query processing environment change, there may be a need to change the VCP scheme and to reorganize the vertical class fragments. Our viewpoint is that, as we are using a cost-based approach, we are in a better position than an affinity-based approach. We can use the cost model to estimate the costs of reorganizing towards a new VCP scheme, or to estimate the extra processing cost if we continue to use the current VCP scheme. As affinity-based approach does not guarantee to generate the cost optimal VCP scheme and hence will not be too useful in predicting the cost for reorganization or the extra processing cost if we continue using the current VCP scheme. From the above discussion, we conclude that VCP is more suitable to be applied to a production type OODB system\(^1\) with rather static database and query processing characteristics. For an evolving type of OODB system\(^2\) with highly dynamic database and query processing characteristics, VCP can still be applied, but we need extra processing to monitor the need for reorganizing and to perform the reorganization of the existing VCP scheme.

Concerning the incorporation of our VCP technique into query processing in an Object Oriented Databases Management System (OODBMS), as a cost model for query processing is an important component in the optimizer of an OODBMS, and we are using a cost-based approach in our VCP technique, therefore, we are in a better position to incorporate our VCP technique into query processing in an OODBMS than the affinity-based approach.

---

1. A production type of OODB system is a system with predefined query processing requirements. Such a system is usually used in daily operations of the organization.

2. An evolving type of OODB system is a system with evolving/dynamic query processing requirements. Such a system is usually used in handling ad hoc requests in the organization.
Chapter 4

Method Induced Vertical Class Partitioning

In object oriented databases (OODBs), methods encapsulated in a class typically access a few, but not all the instance variables defined in the class. It may thus be preferable to vertically partition the class for reducing irrelevant data (instance variables) accessed by the methods. Our prior work [30,31] has shown that vertical class partitioning can result in a substantial decrease in total number of disk accesses incurred for executing a set of applications, but coming up with an optimal vertical class partitioning scheme is an NP-complete problem [68]. In this chapter and in [29,34], we present two algorithms for deriving optimal and near-optimal vertical class partitioning schemes. The cost-driven algorithm (CDA) provides the optimal vertical class partitioning schemes by enumerating, exhaustively, all the schemes and calculating the number of disk accesses required to execute a set of applications. For this, a cost model for executing a set of methods in an OODB system is developed. Since exhaustive enumeration is costly and only works for classes with a small number of instance variables, a hill-climbing heuristic algorithm (HCHA) is developed, which takes the solution provided by the affinity-based algorithm and improves it, thereby further reducing the total number of disk accesses incurred. We show that the HCHA algorithm provides a reasonable near-optimal vertical class partitioning scheme for executing a given set of methods.

The rest of this chapter is organized as follows: Section 4.1 presents the characteristic and specification of the method. Section 4.2 presents an analytical cost model for method-induced vertical partitioning. Section 4.3 presents two different vertical partitioning algorithms. Section 4.4 presents experimental results on these algorithms. A summary is given in section 4.5.

4.1 Method Characteristics and Specification

To lay down a foundation for our subsequent studies, we introduce in this section the concept of
method dependency graph, and describe a general model for methods with an emphasis on the different types of method invocations.

4.1.1 Method Dependency Graph

Method dependency graph (MDG) - MDG is a graphical representation of a complex method that calls/invokes other methods [50]. Each method dependency graph represents the behavior of a complex method accessing an object-based instance variable (OBIV). An MDG has a set of nodes and a set of directed edges. The nodes represent the methods that are invoked by the complex method, and the edges denote the sequence in which the methods are invoked. Since a complex method can invoke other methods (both simple and complex), a complex method can be naturally represented by a set of MDGs. An example MDG is illustrated in Figure 30.

![Figure 30: MDG of an example method ma()](image)

4.1.2 Method Execution Semantics

In most of the previous research, methods are treated as "black boxes" and are neglected from cost analysis. Very little study has been conducted on the cost model for general method execution, partly due to the complexity of the detailed semantics of a general method. Nevertheless, an analytical cost model for method execution in OODBs is a relevant problem to class partitioning. We tackle this problem by studying the cost relationship between different subprograms defined in a method. As many query languages do not (yet) support recursion, we focus on non-recursive methods in this thesis.

The following different types of method invocation are distinguished: (1) simple invocation, (2) conditional invocation, and (3) repeated invocation. We discuss the different types of method invocation below, the purpose of which is to come up with the basic cost formula for each type. For expository purpose, the discussion is based on the three example methods illustrated in Figure 31.
Method invocation dependency (MID) terms

In Figure 31(a), \( \text{ma()} \) defined in class \( A \) is a complex method invoking methods \( mb1() \) and \( mb2() \) in sequence. The two OBIVs \( o1 \) and \( o2 \) are defined in class \( A \), with the domain \( A.o1 \) being the class \( B1 \) and \( A.o2 \) being class \( B2 \). We thus introduce the term of method invocation dependency (MID): \( \text{ma}(o1 \rightarrow mb1, o2 \rightarrow mb2) \), which captures the MDG information of \( \text{ma()} \) in that the methods \( mb1() \) and \( mb2() \) are called directly by method \( \text{ma}() \) (i.e., they are inside the \( "< >" \) which contains the methods called by \( \text{ma}() \)). The notation \( (o1 \rightarrow mb1, o2 \rightarrow mb2) \) implies that methods \( mb1() \) and \( mb2() \) are executed one after the other in sequence, and the notation \( o1 \rightarrow mb1 \) implies that sub-method \( mb1() \) in the MDG is invoked by \( \text{ma()} \) through an object-based instance variable \( o1 \). The IO cost of method \( \text{ma()} \) is calculated by a summation of the IO cost of executing the two sub-methods \( mb1() \) and \( mb2() \), plus the IO cost of executing the method \( \text{ma}() \) without regard to the cost of executing \( mb1() \) and \( mb2() \) (as they are already considered). Figure 31(a) also shows the resultant cost formula in this case. The details of \( \text{IOC} \) and \( \text{Cost} \) will be described in section 4.1.3 and section 4.2.2, respectively.

![Diagram](image)

Figure 31: MDG, sample code and cost formula for (a) simple method invocation, (b) conditional method invocation, and (c) repeated method invocation.

In Figure 31(b), the complex method \( \text{ma()} \) is assumed to invoke methods \( mb1() \) and/or \( mb2() \) conditionally. That is, the method \( \text{ma}() \) branches out to method \( mb1() \) or \( mb2() \). The arc connecting branches in the MDG represents a conditional execution. Furthermore, the method \( mb1() \) is to be invoked with relative frequency \( f1 \) and the method \( mb2() \) to be invoked with relative frequency \( f2 \). Note that per method invocation of \( \text{ma}() \), both of its sub-methods may be executed inclusively or only one of them is actually executed. We therefore introduce a new MID term:
ma([o1/f1 \to mb1|o2/f2 \to mb2]), which means that method \textit{ma()} will execute either method \textit{mb1()} (with frequency \textit{f1}) and/or method \textit{mb2()} (with frequency \textit{f2}); the "\text{"}" inside the "[ ]" means a conditional execution between the two (or more) methods. The IO cost of method \textit{ma()} is calculated by the cost formula as shown in the lower portion of Figure 31(b).

In Figure 31(c), the complex method \textit{ma()} repeatedly invokes method \textit{mb()} with a repeating factor \textit{r} \(^1\) (i.e., \textit{mb()} is executed repeatedly for \textit{r} times). We introduce one more MID term: \textit{ma(\text{"\text{"}r \to mb\text{"}})}. This term depicts that method \textit{ma()} will execute method \textit{mb()} repeatedly (with a repeating factor \textit{r}); the "\text{"\text{"}"} inside the "\text{"\text{"}"} < >" means a repeated execution. The IO cost of method \textit{ma()} is as shown in Figure 31(c), where the cost of method \textit{mb()} is multiplied by the repeating factor \textit{r}. We also assume that the OODBMS keeps track of the repeating factor values. The formula is based on the assumption that we do not have enough main memory buffers, the objects required by \textit{mb()} are repeatedly loaded into the main memory each time they are needed. If we have enough main memory buffers to store all the objects required by \textit{mb()} then the repeating factor \textit{r} will disappear in the cost formula as a consequence.

4.1.3 Method Execution Cost Model
Let \textit{m\langle q_1, q_2, \ldots, q_n \rangle} be the MDG representation of a method \textit{m()}, where \textit{q_1, q_2, \ldots, q_n} are the MID terms (see section 4.1.2), then the cost of executing \textit{m()} is similar to that of [40], that is:

If \textit{q_i} is of form \textit{o_i \rightarrow m_i}, then
\[
\text{Cost}(q_i) = \text{Cost}(m_i).
\]

If \textit{q_i} is of form \textit{\{o_j/f_j \rightarrow m_j\} \ldots |o_k/f_k \rightarrow m_k\}} then
\[
\text{Cost}(q_i) = \frac{f_j}{f_j + \ldots + f_k} \text{Cost}(m_j) + \ldots + \frac{f_k}{f_j + \ldots + f_k} \text{Cost}(m_k).
\]

If \textit{q_i} is of form \textit{o_i^*r_i \rightarrow m_i}, then
\[
\text{Cost}(q_i) = r_i \ast \text{Cost}(m_i).
\]

Similarly \text{Cost}(m_i), \text{Cost}(m_j) and \text{Cost}(m_k) are recursively defined. In conclusion, a nested method invocation can be represented as a combination of the above types of method invocation. We illustrate method execution cost calculation by means of an example.

\[1.\text{ We assume that the underlying OODBMS can monitor the method execution to keep track of these execution frequency values (whenever required by the database administrator). This leads to extra overhead to the OODBMS but will be amortized by the higher-quality cost estimation produced.}\]
Example 1
Consider the method \( ma() \) as shown in Figure 30, it can be represented by using MID terms as follows:

\[ ma(o_1 \rightarrow mb_1, [o_2f/1 \rightarrow mb_2|o_3f/2 \rightarrow mb_3], o_4*r_1 \rightarrow mb_4) \]

The cost of executing \( ma() \) is calculated as follows:

\[
\text{Cost}(ma(o_1 \rightarrow mb_1, [o_2f/1 \rightarrow mb_2|o_3f/2 \rightarrow mb_3], o_4*r_1 \rightarrow mb_4)) = \text{IOCost}(ma) + \\
\frac{f_1}{f_1 + f_2} \text{Cost}(mb_1(o_5f/3 \rightarrow mc_1|o_6f/4 \rightarrow mc_2|o_7f/5 \rightarrow mc_3)) + \\
\frac{f_2}{f_1 + f_2} \text{Cost}(mb_3) + \\
r_1 \text{Cost}(mb_4(o_9 \rightarrow mc_5)).
\]

In the above formula,

\[
\text{Cost}(mb_1(o_5f/3 \rightarrow mc_1|o_6f/4 \rightarrow mc_2|o_7f/5 \rightarrow mc_3)) = \text{IOCost}(mb_1) + \\
\frac{f_3}{f_3 + f_4 + f_5} \text{Cost}(mc_1) + \\
\frac{f_4}{f_3 + f_4 + f_5} \text{Cost}(mc_2) + \\
\frac{f_5}{f_3 + f_4 + f_5} \text{Cost}(mc_3),
\]

\[
\text{Cost}(mb_2(o_8*r_2 \rightarrow mc_4)) = \text{IOCost}(mb_2) + r_2 \text{Cost}(mc_4),
\]

and

\[
\text{Cost}(mb_4(o_9 \rightarrow mc_5)) = \text{IOCost}(mb_4) + \text{Cost}(mc_5),
\]

with

\[
\text{Cost}(mb_3) = \text{IOCost}(mb_3), \quad \text{Cost}(mc_1) = \text{IOCost}(mc_1),
\]

\[
\text{Cost}(mc_2) = \text{IOCost}(mc_2), \quad \text{Cost}(mc_3) = \text{IOCost}(mc_3),
\]

\[
\text{Cost}(mc_4) = \text{IOCost}(mc_4), \quad \text{and} \quad \text{Cost}(mc_5) = \text{IOCost}(mc_5).
\]

\[\blacksquare\]

4.1.4 Estimation of Parameter Values
To enable us to calculate the cost of executing a complex method, we need to have estimates on the following method invocation parameters: (i) the relative frequencies between different sub-methods in conditional invocation; and (ii) the repeating factor for repeated invocation. These estimates can be obtained from any one or all of the following sources:

- (a) extra counter variables may be attached to each method to obtain the method invocation parameter values;
- (b) the underlying OODBMS monitors and keeps track of statistics on these method invocation parameter values.
- (c) the method designer provides a preliminary estimate;

With extra overhead, accurate parameter values can also be obtained from (a) and (b). The extra overhead of monitoring/keeping track of the query/method execution in (a) and (b) is amortized by the performance gain after we applied the method-induced VCP technique. In (c), the parameter
values are estimated by the VCP designer; the values will not be too accurate, this will affect the utility of VCP. However, our VCP cost model provide guidelines for choosing the initial VCP scheme. And the VCP scheme can be further refined when more accurate parameter values are obtained in the future.

### 4.2 Cost model

In this section, we present an analytical cost model for method-induced vertical partitioning in OODBs.

**Table 8: Cost model parameters for MI-VCP**

<table>
<thead>
<tr>
<th>Category</th>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Database</strong></td>
<td>$|C_{i, k}|$</td>
<td>cardinality of class collection $C_{i, k}$ (i.e., $k$ th subclass of $i$ th class along the class composition hierarchy)</td>
</tr>
<tr>
<td></td>
<td>$</td>
<td>C_{i, k}</td>
</tr>
<tr>
<td></td>
<td>$SC_{i, k}$</td>
<td>size of object (in unit of byte) in class $C_{i, k}$</td>
</tr>
<tr>
<td></td>
<td>$q_{i}$</td>
<td>number of subclasses in the class inheritance hierarchy rooted by class $C_{i}$</td>
</tr>
<tr>
<td></td>
<td>$fan_{i-1, i, j, k}$</td>
<td>fan-out for the class composition hierarchy from $j$ th subclass of class $C_{i-1}$ to the $k$ th subclass of class $C_{i}$</td>
</tr>
<tr>
<td></td>
<td>$n$</td>
<td>path length of the MDG path, i.e., the number of classes along the MDG path in the class composition hierarchy</td>
</tr>
<tr>
<td></td>
<td>$h$</td>
<td>path length of the hidden path</td>
</tr>
<tr>
<td></td>
<td>$p$</td>
<td>path length of the parameter path</td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>B*-tree index average fan-out</td>
</tr>
<tr>
<td></td>
<td>$PS$</td>
<td>page size of the file system (in unit of byte)</td>
</tr>
<tr>
<td><strong>Method</strong></td>
<td>$M_{i, j, k}$</td>
<td>a binary variable, it is of value 1 if the method accesses $j$ th vertical fragment of the $k$ th subclass in class $C_{i}$; 0 otherwise</td>
</tr>
<tr>
<td></td>
<td>$Sproj_{i, k}$</td>
<td>length of output result that is within $k$ th subclass in class $C_{i}$</td>
</tr>
<tr>
<td></td>
<td>$Sproj_{i, j, k}$</td>
<td>length of output result that is within $j$ th fragment of the $k$ th subclass in class $C_{i}$</td>
</tr>
<tr>
<td></td>
<td>$ref_{i, k}$</td>
<td>number of object references for $k$ th subclass in class $C_{i}$ during the path expression evaluation process along the class composition hierarchy in the MDG path</td>
</tr>
<tr>
<td></td>
<td>$SEL_{i}$</td>
<td>selectivity of the method's predicate on class $C_{i}$</td>
</tr>
<tr>
<td><strong>Vertical</strong></td>
<td>$m_{i}$</td>
<td>number of fragments in class $C_{i}$</td>
</tr>
<tr>
<td><strong>Partitioning</strong></td>
<td>$SC_{i, j, k}$</td>
<td>size of $j$ th fragment in $k$ th subclass in class $C_{i}$</td>
</tr>
<tr>
<td></td>
<td>$</td>
<td>C_{i, j, k}</td>
</tr>
<tr>
<td></td>
<td>$VP_{i}$</td>
<td>a binary variable, it is of value 1 if class/subclasses of $C_{i}$ is vertically partitioned. 0 otherwise</td>
</tr>
<tr>
<td></td>
<td>$</td>
<td>CO_{i, k}</td>
</tr>
</tbody>
</table>

For the hidden path, the following parameters are similarly defined: $[H_{i, k}]$, $[H_{i, k}, SH_{i, k}, hq_{i}, Hfan_{i-1, i, j, k}, HM_{i, j, k}, Href_{i, k}, HSEL_{i}, hm_{i}, SH_{i, k}V_{j}, H_{i, k}V_{j}, HVP_{i}, HCO_{i, k}]$. For the parameter path, the following parameters are similarly defined: $[P_{i, k}], [P_{i, k}, SP_{i, k}, pq_{i}, Pfan_{i-1, i, j, k}, PM_{i, j, k}, Pref_{i, k}, PSEL_{i}, pm_{i}, SP_{i, k}V_{j}, [P_{i, k}V_{j}], PVP_{i}, [PCO_{i, k}]]$. 

80
4.2.1 Basic concepts

Similar to other related work (e.g., [37]) on cost models in OODBs, we have the cost model parameters as shown in Table 8. Some of the parameters can readily be obtained from the system catalog (like sizes of instance variables), while others require the OODBMS to keep track of their values, such as the fan-outs between classes in the class composition hierarchy.

To facilitate the building of the cost model, we define the following types of path expressions (with the last two being adopted from [77]):

- **MDG path expression** - a path expression obtained from the MDG. An example is $A \cdot o_1 \cdot o_5$ of Figure 30. Note that an MDG can contain more than one MDG path expression.

- **Parameter path expression** - a path expression originating from parameter objects of a method.

- **Hidden path expression** - all path expressions that are not MDG or parameter path expressions.

A particular MDG path (i.e., one of the several possible MDG paths in an MDG) with path length $n$ is of the general form $c_1 \cdot o_2 \cdot o_3 \cdot \ldots \cdot o_{n-1} \cdot o_n$, where $c_1$ is the root class of the MDG, and $o_i \in c_{i-1}$, i.e., $o_i$ is the OBIV defined in class $c_{i-1}$ with a domain of class $c_i$. An example schema of an MDG path with a class inheritance hierarchy is shown in Figure 32. For convenience, we denote the $k$ th subclass of a class inheritance hierarchy rooted at class $c_i$ by the notation $c_{i,k}$, where $k$ ranges over 1 through $q_i$ (which is the total number of subclasses of class $c_i$). To make the cost formulae more compact, we optionally denote the root class $c_i$ as $c_{i,0}$. We further denote the $j$ th vertical fragment of the $k$ th subclass of class $c_i$ by $c_{i,k}V_j$, where $j$ ranges over 1 through $m_i$ (which is the total number of fragments for every class in the class inheritance hierarchy rooted at class $c_i$).

We assume after vertical partitioning, all the class/subclasses of the whole class inheritance hierarchy are partitioned into the same number of fragments. An example schema for an MDG path with vertical partitioning is shown in Figure 33. Similar notation is also used for hidden and parameter paths.

![Diagram](image)

Figure 32: Example schema for a MDG path in Unpartitioned case
As in Chapter 3, the total number of pages occupied by a class collection $C$ with object size $SC$ is given by: $|C| = \left\lceil \frac{|C| \times SC}{PS} \right\rceil$. In evaluating a predicate, we estimate the number of page accesses to a class collection during path expression traversal. We still use the $Y$ function (i.e., the extended Yao function [78]).

### 4.2.2 IO Cost formulae for method execution

In [37], the query processing cost along a path expression (either MDG, parameter or hidden path expression) has the following three cost components:

$$IOPathExpression = IOLoad + IOEval + IOBuild$$

where $IOLoad$ is the number of disk IOs to load in the whole root class collection, $IOEval$ is the total number of disk IOs to evaluate the predicate by traversing through the different classes along the path expression, and $IOBuild$ is the number of disk IOs to build the results. In [30], we extend [37] to query processing in both unpartitioned and vertically partitioned OODBs.

In this thesis, these three cost components are further extended to the different path expressions (namely MDG, parameter and hidden path expressions) in a complex method. As described in section 4.1.3, the cost of evaluating a complex method $m$ is calculated by the general cost formula of $Cost(m)$ which is dependent on $IOCost$ functions of each (sub)method involved. We thus concentrate here on the cost formulae of $IOCost(m)$ for a complex method $m$ (defined in a class $c_i$) without regard to other method invocations. Similar to $IOPathExpression$, the $IOCost(m)$ includes the following components:

$$IOCost(m) = IOLoad + IOMDGPath + IOBuild + \sum IOHiddenPaths + \sum IOParamPaths$$
where (1) IOLoad is the number of disk IOs to load in the whole root class collection (with \( C_i \) being the root class) of an MDG, (2) IOMDGPPath is the number of disk IOs to evaluate "one node" (not all the nodes) in an MDG path expression which is from a previous class \( C_{i-1} \) to the current class \( C_i \) (if \( C_i \) is not the root class in the MDG)\(^1\), (3) IObuild is the number of disk IOs to build the results, (4) IOHiddenPath is the number of disk IOs to process the whole hidden path expression\(^2\), and (5) IOParamPath is the number of disk IOs to process the whole parameter path expression\(^3\).

After this we use the formulae developed in section 4.1.3 to calculate the total cost of invoking complex method \( m \).

We note that sequential scan and index scan are the two major strategies used for scanning a class collection. As mentioned before, the objective of using an index is to attain faster object access, while the objective of using vertical partitioning is to reduce irrelevant data access. In the general case, since not every instance variable defined in a class has an index associated with it, the objects in the class collection are assumed to be accessed sequentially in order to have a uniform comparison of experimental results.

The cost formulae for sequential scan are summarized in Table 9, while the detailed derivation is similar to that of Chapter 3.

For methods that use index scan to a class collection, the cost model formulae differ only in the IOLoad, IOHiddenLoad and IOParamLoad components. For example, according to [37], for non-clustered index implemented as a B\(^+\)-tree with average fan-out of \( b \), the IOLoad for the unpartitioned case is

\[
\log_b \left( \text{SEL}_1 \times \sum_{k=0}^{q_1} \frac{||C_{1,k}||}{\text{NP}_{1,k}} \right) + \sum_{k=0}^{q_1} Y(||C_{1,k}||, ||C_{1,k}||, ref_{1,k}),
\]

and the IOLoad for the vertical partitioned case is:

\[
\log_b \left( \text{SEL}_1 \times \sum_{k=0}^{q_1} \frac{||C_{1,k}||}{\text{NP}_{1,k}} \right) + \sum_{k=0}^{q_1} Y(||C_{1,k}||, ||C_{1,k}||, ref_{1,k}) + \sum_{j=1}^{m_1} \sum_{k=0}^{q_1} Y(||C_{1,k}||, ||C_{1,k}V_j||, ref_{1,k}).
\]

The IOHiddenLoad and IOParamLoad can be similarly defined.

---

1. As we are calculating the method execution cost for the current class, we only need to consider the traversal cost from the previous class towards the current class.

2. Note that there may be multiple hidden path expressions within a method and IOHiddenPath=IOHiddenLoad+IOHiddenEval. As the results are from the classes along the MDG path expression, hence IOHiddenBuild is zero.

3. Also note that there may be multiple parameter path expressions within a method and IOParamPath=IOParamLoad+IOParamEval. As the results are from the classes along the MDG path expression, hence IOParamBuild is also zero.
### Table 9: Cost model formulae for MI-VCP

<table>
<thead>
<tr>
<th></th>
<th>Unpartitioned</th>
<th>Vertical Partitioned</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IOLoad</strong></td>
<td>( \sum_{k=0}^{q_1}</td>
<td>C_{1,k}</td>
</tr>
<tr>
<td><strong>IOMDPath</strong></td>
<td>( \sum_{k=0}^{q_1} Y(</td>
<td></td>
</tr>
<tr>
<td><strong>IOBUILD</strong></td>
<td>( \sum_{k=0}^{q_1} \frac{SEL_{i} \times</td>
<td></td>
</tr>
<tr>
<td><strong>IOHiddenLoad</strong></td>
<td>( \sum_{k=0}^{kq}</td>
<td>H_{1,k}</td>
</tr>
<tr>
<td><strong>IOHiddenEval</strong></td>
<td>( \sum_{i=2}^{q_i} Y(</td>
<td></td>
</tr>
<tr>
<td><strong>IOParamLoad</strong></td>
<td>( \sum_{k=0}^{pq}</td>
<td>P_{1,k}</td>
</tr>
<tr>
<td><strong>IOParamEval</strong></td>
<td>( \sum_{i=2}^{pq} Y(</td>
<td></td>
</tr>
</tbody>
</table>

### 4.3 Vertical partitioning algorithms

Given an OODB schema and a set of methods with specified execution characteristics (MDGs and MID terms), vertical partitioning algorithms can be devised to generate optimal or near-optimal partitioning schemes. The problem of coming up with an optimal vertical class partitioning scheme is NP-complete [68], and hence heuristic algorithms need to be designed to come up with near-op-
timal vertical class partitioning schemes. Two main heuristics, namely, affinity based [62] and cost driven [22] have been proposed in prior research. In this section, we present two algorithms for vertical class partitioning: a cost-driven algorithm which is a pure cost-based approach, and a hill-climbing heuristic algorithm which is based on a combination of affinity-based and cost-based approaches.

### 4.3.1 Cost-Driven Algorithm (CDA)

The cost-driven algorithm (CDA) uses the cost model for method execution. By exhaustively searching all the partitioning schemes, it finds the optimal vertical partitioning scheme that minimizes the cost. The algorithm consists of 2 stages. In the first stage, we perform a detailed analysis on the cost relationship and the different cost component distributions among the different classes for every method defined in the schema. In the second stage, we make use of the cost information obtained to derive the optimal vertical partitioning scheme. We begin the first stage by studying the cost relationship within the MDG for each method defined in the schema. Different types of method invocation are identified, which include simple, conditional and repeated method invocations. (Again, we assume that the underlying OODBMS monitors the method execution and keeps track of the relative frequencies between conditional method invocations and the repeating factors under repeated invocations.) We then study the detailed cost components within each class for each method defined in the schema; these components include the IO costs involved in the different stages of method execution: IOLoad, IOMDGPPath, IOBuild, IOHiddenPath and IOParamPath. We treat polymorphic/redefined methods as different methods during the analysis. At the end of the first stage, we obtain detailed cost formulae and cost distribution among the different classes for each method defined in the schema. The above information is used in the find_method_execution_cost_within_that_class procedure of the CDA algorithm to find the method execution cost. In this respect, CDA is more suitable for production OODBs (with predefined method execution characteristics) because all the above information can be collected and stored in the system catalog.

In the second stage, we make use of the cost information to derive the optimal vertical partitioning scheme. For each class in the schema, we enumerate all possible vertical partitioning schemes of that class, and for each possible vertical partitioning scheme Si, we calculate the total IO required (which takes into account the cost relationship and the cost components) for each method, and sum up the grand total IO cost required for all the methods under the scheme Si. After enumerating all possible vertical partitioning schemes, the overall minimum cost vertical partitioning scheme for a particular class can be obtained. The algorithm is as follows:

85
/* first stage */
Step 1: Perform detailed analysis on cost relationship and cost components
/* second stage, perform exhaustive enumeration on every class */
Step 2: For every class in the schema do
Step 3: \( \text{min} := +\infty \)
Step 4: For every possible vertical partitioning scheme in that class do
Step 5: \( \text{total}_\text{IO} := 0 \)
Step 6: For every method do
Step 7: \( \text{IOCost} := \text{find\_method\_execution\_cost\_within\_that\_class} \)
Step 8: \( \text{total}_\text{IO} := \text{total}_\text{IO} + \text{IOCost} \)
Step 9: EndFor
Step 10: If \( \text{total}_\text{IO} < \text{min} \)
Step 11: \( \text{min} := \text{total}_\text{IO} \)
Step 12: \( \text{optimal\_configuration} := \text{current\_configuration} \)
Step 13: EndIf
Step 14: EndFor
Step 15: EndFor

Figure 34: Cost-Driven Algorithm

Although our cost model is general enough to cater for all the cases, to make the application environment comparable and applicable with the affinity-based approach, we make the following simplifying assumptions on the methods in the analytical experiments (cf. section 4.4): (a) as the affinity-based approach does not model the concept of hidden and parameter paths, we leave out methods with hidden and parameter paths; (b) when calculating the number of object references along an MDG path expression, the following formula is used:

\[
\text{ref}_{i,k} = \left( \sum_{j=0}^{q_{i-1}} \text{ref}_{i-1,j} \times \text{fan}_{i-1,j,i,k} \right) \times \text{SEL}_{i-1}, \text{with } 1 < i \leq n, \ 0 \leq k \leq q_{i}, \text{ where } \text{SEL}_{i-1} \text{ is set to one (i.e., }
\]

all the object instances in the current class collection are scanned during the traversal of the MDG path expression, which corresponds to the worst case scenario in the MDG path expression traversal).

### 4.3.2 Hill-Climbing Heuristic Algorithm (HCHA)

Our second algorithm combines the well known affinity-based vertical algorithm (viz., the graph-based algorithm described in [64]) with the cost-based approach, so as to achieve a reasonable near-optimal result in an efficient manner. We start with a recap of the affinity-based algorithm first.

**Pure affinity-based vertical partitioning algorithm**

As described in [64], the affinity-based algorithm starts from the instance variable affinity matrix which is generated from the instance variable usage matrix. An *instance variable usage matrix* (IVUM) represents the use of instance variables. Each row in the matrix refers to a method; a "1" entry in a column indicates that the method accesses the corresponding instance variable. An IVUM element \( u_{i,j} \) is set to "1" if the \( t \) th method accesses the \( i \) th instance variable; "0" otherwise. Figure 35(a) shows an example IVUM.
Based on IVUM, an *instance variable affinity matrix* (IVAM) element is defined as
\[ a_{i,j} = \sum_{t \in \text{Methods}} (u_{t,i} \wedge u_{t,j}) \times \text{freq}_t, \]
where the summation is over all the methods, the "\( \wedge \)" is the logical AND operator and \( \text{freq}_t \) is the frequency of method execution of the \( t \)th method. Each IVAM matrix element measures the strength of an "imaginary bond" \([64]\) between the two instance variables when they are accessed together by methods. An example IVAM (which corresponds to the above IVUM) is shown in Figure 35(b).

As in [64], the algorithm starts with the instance variable affinity matrix by considering it as a complete graph. It then forms a linearly connected spanning tree and generates all meaningful vertical fragments simultaneously by considering a cycle as a fragment. This algorithm is based on the fact that all pairs of attributes in a fragment have high intra-fragment affinity and low inter-fragment affinity. Compared with previous vertical partitioning algorithms, this algorithm has computational superiority and is of complexity \( O(n^2) \) (where \( n \) is the number of instance variables in the class).

In our approach, we first generate the instance variable usage matrix by analysing the method definition and then apply the graph-based algorithm [64] to generate an optimal affinity-based vertical partitioning scheme. Concerning the instance variable usage matrix, we do not need to separate the instance variables into groups (such as one group per class), as the graph-based algorithm is general enough to cater for all the instance variables in the whole schema in one shot. The [64] algorithm is included in Appendix A for reference. Note that the affinity-based algorithm is more suitable for determining an initial vertical class partitioning scheme.

**The Hill-Climbing Heuristic Algorithm**

As the exhaustive enumeration strategy used in the cost-driven algorithm (CDA) requires a high computational cost of order \( O(n^n) \) (where \( n \) is the number of instance variables in the class), it is impractical when the total number of instance variables in the schema is very large. On the other hand, the pure affinity-based approach is not as comprehensive as the cost-based approach in modeling important database characteristics, like the sizes of the instance variables. Furthermore, the affinity-based approach cannot yield an analytical cost comparison between different partitioning
schemes, yet such comparison is very crucial in comparing the effectiveness of different vertical partitioning algorithms. Some heuristic approach is therefore needed to tackle the vertical partitioning problem effectively. Here we introduce a hill-climbing heuristic algorithm (HCHA). The HCHA algorithm uses the concept of popularity, and is defined as

**Definition 1**

The *popularity* of a particular instance variable $v_i$ is the sum of the frequencies of the methods (transactions) which access that instance variable.

In HCHA, there are four major elements in the hill-climbing heuristic [65]:

- **initial state:** we use the [64] graph-based algorithm's optimal partitioning scheme as the initial state. This is a good choice because the affinity-based optimal partitioning scheme is closer to the real cost-based optimal partitioning scheme than any random and/or *ad hoc* initial guess;

- **next state:** we shuffle the instance variables in the different fragments in the current partitioning scheme to generate the next state. We have several "move" operations from the current state to the next state, which can involve: (a) migrating an instance variable from one fragment $F_i$ to another fragment $F_j$ (by selecting from $F_i$ the instance variable $v$ so that there is an instance variable $w$ from $F_j$, with $(v, w)$ being of the highest *inter-fragment affinity*), (b) grouping two instance variables from two fragments to form a new fragment, (c) separating one or more instance variables that have the lowest popularity\(^1\) from a fragment to form a new fragment, where the popularity for the $i$th instance variable $v_i$ is defined as the $i$th diagonal element in the instance variable affinity matrix used by the affinity-based algorithm. The intuition is that we should group instance variables which are frequently accessed together, i.e., those instance variables having similar/comparable popularities, to form a fragment, so that the variations of the popularities within the fragment will be small.

- **comparison:** we use the cost-based approach's cost formulae to calculate the cost required for the next partitioning scheme to see if it is of lower cost than the current partitioning scheme; and

- **termination:** we terminate the hill-climbing algorithm in any iteration when we cannot find any partitioning scheme with lower cost than the current partitioning scheme.

---

\(1\). Note that we use popularities to decide if a fragment of a partitioning scheme $S$ should be split. When there is more than one fragment in $S$, we select the fragment $F$ with the largest variation among the popularities of its instance variables.
The HCHA algorithm as shown in Figure 36 is thus a 2-stage process: stage 1 is the graph-based vertical partitioning algorithm [64], and stage 2 is the application of the hill-climbing heuristics.

The procedure \texttt{find\_cost} calculates the total cost of method execution of the input partitioning scheme using the cost-based formulae. There are 4 procedures in the algorithm to find the "next move": (1) \texttt{left\_merge}; (2) \texttt{right\_merge}; (3) \texttt{split\_new} and (4) \texttt{single\_split}. Given the current partitioning scheme (\texttt{curr\_part\_scheme}), procedure \texttt{find\_max\_inter\_fragment\_affinity} finds a pair of instance variables from two different fragments with maximum inter-fragment affinity (by consulting the instance variable affinity matrix produced in stage one). The three procedures \texttt{left\_merge}, \texttt{right\_merge} and \texttt{split\_new} all require the maximum inter-fragment affinity pair. For example, if instance variables \( B \) and \( D \) are of maximum inter-fragment affinity for the current partitioning scheme: \( (A \ B \ C \ D \ E \ F \ G) \), procedures \texttt{left\_merge}, \texttt{right\_merge} and \texttt{split\_new} will yield \( ((A \ B \ C \ D) \ E \ F \ G) \), \( ((A \ C) \ (B \ D \ E) \ F \ G) \) and \( ((A \ C) \ (B \ D) \ E \ F \ G) \), respectively. On the other hand, procedure \texttt{find\_min\_popularity} finds within the fragment the instance variable with minimum popularity but of the largest variation in popularity. For example, if \((100\ 60\ 80\ \ 0\ \ 0\ \ 50\ \ 50)\) is the popularity for a current partitioning scheme \( (A \ B \ C \ D \ E \ (F \ G)) \), the minimum popularity instance variable will then be \( B \) since \( (A \ B \ C) \) has larger variation than \( (F \ G) \). Therefore the next partitioning scheme will become \( ((A \ C) \ (D \ E) \ (F \ G) \ (B)) \) after applying the procedure \texttt{single\_split}.

Note that HCHA will not enter an infinite loop or fail to terminate. This is because not only the total number of different partitioning schemes is finite, but more importantly our \texttt{old\_cost} is ever decreasing, so if the algorithm generates a particular partitioning scheme that has been generated before, this partitioning scheme will not be considered as the next one, since the cost of it cannot be less than itself. Another characteristic of this algorithm is that it either obtains an improved partitioning scheme or terminates at once. As our initial partitioning scheme is the affinity-based optimal partitioning scheme generated from the [64] algorithm, HCHA is guaranteed to always produce a partitioning scheme which is at least as good as that of the [64] algorithm.
/* first stage */
Step 1: Perform [64] graph-based algorithm /* it outputs the affinity-based
optimal partitioning scheme and the instance variable affinity matrix
for later use */

/* second stage */
Step 2: For each class in the schema
Step 3: curr_part_scheme:= affinity-based optimal partitioning scheme
Step 4: old_cost:= find_cost(curr_part_scheme)
Step 5: finished:= false
Step 6: While not finished
Step 7: Perform find_max_inter_fragment_affinity
Step 8: Perform find_min_popularity
Step 9: For next_move in [left_merge, right_merge, split_new, single_split]
Step 10: next_part_scheme:= next_move(curr_part_scheme)
Step 11: new_cost:= find_cost(next_part_scheme)
Step 12: If new_cost<old_cost
Step 13: curr_part_scheme:= next_part_scheme
Step 14: old_cost:= new_cost
Step 15: Goto step 19
Step 16: EndIf
Step 17: EndFor
Step 18: finished:= true /* Cannot find any next state with lower cost */
Step 19: EndWhile
Step 20: EndFor

Figure 36: Hill-Climbing Heuristic Algorithm

4.4 Evaluation of the Vertical Partitioning Algorithms

In this section we present the results of analytical evaluation experiments on the vertical partitioning algorithms (viz., CDA and HCHA) introduced in the previous section. For better comparing and contrasting purposes, we treat here the pure affinity-based algorithm [64] as a separate, additional algorithm for vertical class partitioning. Therefore, the pros and cons of the pure affinity algorithm are also discussed along with that of CDA and HCHA.

4.4.1 OODB Schema

We use the following database schema for our analytical evaluation experiments.

<table>
<thead>
<tr>
<th>Class STAFF</th>
<th>Class PROJECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Properties:</td>
<td>Properties:</td>
</tr>
<tr>
<td>staff_name</td>
<td>name</td>
</tr>
<tr>
<td>basic_salary</td>
<td>supervisor</td>
</tr>
<tr>
<td>Methods:</td>
<td>staff</td>
</tr>
<tr>
<td>staff_name()</td>
<td>pj_loc()</td>
</tr>
<tr>
<td>basic_salary</td>
<td>plan()</td>
</tr>
<tr>
<td></td>
<td>expense()</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>char[40];</td>
<td>char[40];</td>
</tr>
<tr>
<td>int;</td>
<td>STAFF;</td>
</tr>
<tr>
<td></td>
<td>setof STAFF;</td>
</tr>
<tr>
<td></td>
<td>BRANCH;</td>
</tr>
<tr>
<td>int;</td>
<td>char[200];</td>
</tr>
<tr>
<td></td>
<td>setof STAFF;</td>
</tr>
<tr>
<td></td>
<td>BRANCH;</td>
</tr>
<tr>
<td>int;</td>
<td>char[200];</td>
</tr>
<tr>
<td></td>
<td>int;</td>
</tr>
<tr>
<td>Class</td>
<td>Properties</td>
</tr>
<tr>
<td>-------</td>
<td>------------</td>
</tr>
<tr>
<td>ADM_STAFF</td>
<td>fringe_benefit: int;</td>
</tr>
<tr>
<td>TECH_STAFF</td>
<td>ot_hours: int;</td>
</tr>
<tr>
<td>SITE</td>
<td>address: char[120];</td>
</tr>
<tr>
<td>BRANCH</td>
<td>br_name: char[40];</td>
</tr>
<tr>
<td></td>
<td>estate_tax: int;</td>
</tr>
<tr>
<td></td>
<td>br_locs: setof SITE;</td>
</tr>
<tr>
<td></td>
<td>br_staff: setof STAFF;</td>
</tr>
<tr>
<td></td>
<td>function: char[100];</td>
</tr>
<tr>
<td></td>
<td>br_cost(): int;</td>
</tr>
</tbody>
</table>

### 4.4.2 Example methods

In our example, the method definitions are divided into two categories: (1) **elementary instance variable access (EIVA) methods**, and (2) **application methods**. EIVA methods are defined on a one-to-one correspondence with the instance variables of a class. They are used to access the instance variable values, which facilitates the encapsulation properties of the OODB model and the hiding of the implementation details of the instance variables. For example, method `estate_tax()` returns the estate tax of the `BRANCH` class object. Application methods are defined by the application designer to access the database. Usually these methods access multiple instance variables, and may return some results and/or invoke other methods. In our example, we have the following methods belonging to this category:

- `income()` in class `TECH_STAFF` returns the salary as calculated by `basic_salary + ot_hours * ot_hourly_rate`;

- `income()` in class `ADM_STAFF` returns the salary as calculated by `basic_salary + fringe_benefit`; (Note that `TECH_STAFF` and `ADM_STAFF` are subclasses of `STAFF`, and they have different definitions for the `income()` method.)

- `rent()` in class `SITE` returns the rent as calculated by `rent_per_sqft * size`;

- `br_cost()` in class `BRANCH` returns the cost as calculated by adding `estate_tax` with the sum of rents of all `br_locs` sites;

- `expense()` in class `PROJECT` returns the expense as calculated by adding salary of supervisor
with the sum of salary of all staff and the \textit{br\_cost} of \textit{pj\_loc} branch. The MDG of this complex method is illustrated in Figure 37.

![MDG of complex method expense()](image)

**Figure 37: MDG of complex method expense()**

### 4.4.3 Experimental results

Our analytical evaluations are performed using the cost model of section 4.2 upon the three vertical partitioning algorithms through two different sets of methods: (i) one with rather even distribution of instance variable accesses, which means the expected performance gain of using vertical partitioning will not be substantial; (ii) the other one with skewed instance variable accesses, which means some instance variables are heavily accessed and the expected performance gain of using vertical partitioning will be substantial. First we present the results on the HCHA approach. To be concise, the results for the \textbf{STAFF} class actually represent the results for the whole \textbf{STAFF} class inheritance hierarchy.

#### Case (i) - even distribution of instance variable accesses:

<table>
<thead>
<tr>
<th>Case (i)</th>
<th># of iterations</th>
<th># of partitioning schemes checked</th>
<th>IO cost of initial partitioning scheme by HCHA</th>
<th>IO cost of final partitioning scheme by HCHA</th>
<th>Optimal IO cost as predicted by CDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROJECT</td>
<td>6</td>
<td>17</td>
<td>14920</td>
<td>9520</td>
<td>9520</td>
</tr>
<tr>
<td>STAFF</td>
<td>2</td>
<td>8</td>
<td>27790</td>
<td>27740</td>
<td>27740</td>
</tr>
<tr>
<td>BRANCH</td>
<td>6</td>
<td>19</td>
<td>13590</td>
<td>9300</td>
<td>9300</td>
</tr>
<tr>
<td>SITE</td>
<td>5</td>
<td>15</td>
<td>22830</td>
<td>17440</td>
<td>17440</td>
</tr>
</tbody>
</table>

As shown in the above table, all four classes’ costs based on the HCHA approach attain the optimal IO costs as predicted by the pure cost-based approach. The conclusion is that HCHA’s predicting power of the optimal partitioning scheme is good. Also note that the total number of partitioning
schemes checked in the CDA approach using the exhaustive enumeration strategy is 52\(^1\) for each of the four classes, but the number of partitioning schemes checked in the HCHA case is much lower than the CDA approach. Thus, a companion conclusion is that the HCHA approach is more efficient than the CDA approach in the even distribution case.

**Case (ii) - skewed distribution of instance variable accesses:**

<table>
<thead>
<tr>
<th>Class</th>
<th># of iterations</th>
<th># of partitioning schemes checked</th>
<th>IO cost of initial partitioning scheme by HCHA</th>
<th>IO cost of final partitioning scheme by HCHA</th>
<th>Optimal IO cost as predicted by CDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROJECT</td>
<td>5</td>
<td>13</td>
<td>5280</td>
<td>4600</td>
<td>4520</td>
</tr>
<tr>
<td>STAFF</td>
<td>2</td>
<td>5</td>
<td>2560</td>
<td>2500</td>
<td>2500</td>
</tr>
<tr>
<td>BRANCH</td>
<td>4</td>
<td>10</td>
<td>2080</td>
<td>1820</td>
<td>1820</td>
</tr>
<tr>
<td>SITE</td>
<td>4</td>
<td>10</td>
<td>2700</td>
<td>2280</td>
<td>2280</td>
</tr>
</tbody>
</table>

In this case, the distribution of instance variable accesses is skewed, meaning that some of the instance variables are heavily accessed and some of the instance variables are seldom accessed by the methods\(^2\). As shown in the above table, the HCHA approach attains the optimal IO costs for three out of the four classes as predicted by the CDA approach. For the missed case (i.e., PROJECT class), the IO cost of the final partitioning scheme is 4600, which means only a 2\% error (since 4600/4520=1.02). The conclusion is that HCHA’s predicting power of the optimal partitioning scheme is still good. Also, we can see that the number of partitioning schemes checked in this case is much lower than the CDA’s exhaustive enumeration strategy (which requires 52). This again confirms that HCHA is much more efficient than the CDA even for the skewed case.

---


2. The instance variables that are heavily accessed in our examples are: name, supervisor (in class PROJECT), basic_salary (in class STAFF), br_name, br_locs (in class BRANCH), and rent_per_sqft, size (in class SITE).
The results of all the three algorithms are now summarized and compared as follows:

**Computation cost required by the algorithms:** As CDA uses an exhaustive enumeration strategy, the computation cost is very high for a large number of instance variables. For the pure affinity-based approach, the graph-based algorithm is very efficient and the computation cost of the algorithm is low. The computation cost for the HCHA is always between the CDA and the pure affinity-based approach; also the performance is dependent upon the instance variable access patterns: if the initial guess from the pure affinity-based approach is already close to optimal, then the extra computation cost to obtain the optimal scheme will be very low.

**I/O cost gain:** The performance metric is the Normalized IO, which is defined as the cost ratio between vertical partitioned class and unpartitioned class. In case of even distribution of instance variable accesses, Figure 38 illustrates that the Normalized IO (in the Total column) of the HCHA
approach has the same performance as the CDA approach, which is 1.0 - 0.47 = 53% better than the unpartitioned case. On the other hand, the pure affinity-based approach is 1.0 - 0.58 = 42% better than the unpartitioned case. Therefore, in the case of even distribution of instance variable accesses, the CDA and HCHA approaches are the best and they are (0.53-0.42)/0.53 = 21% better than the pure affinity-based approach. For skewed distribution of instance variable accesses, Figure 39 shows that the Normalized IO (in the Total column) of the HCHA approach has similar performance as CDA, namely they are 1.0 - 0.29 = 71% better than the unpartitioned case (which is quite a substantial gain); the pure affinity-based approach is 1.0 - 0.33 = 67% better than the unpartitioned case. Therefore, in the case of skewed instance variable accesses, the CDA and HCHA approaches are still the best, even though they are only (0.71-0.67)/0.71 = 6% better than the pure affinity-based approach in this skewed case. We note that as the instance variable accesses become more skewed, the differences in the ability to identify the optimal partitioning scheme for the different approaches becomes smaller.

Comparing the pros and cons of the three algorithms:

(1) **Cost-driven algorithm**: (a) it has the advantage of obtaining the optimal partitioning scheme; (b) it has a high computation cost if the number of instance variables is large; and (c) it is most suitable for a production OODB system.

(2) **Pure affinity-based algorithm**: (a) it has the advantage of low computation cost even with a large number of instance variables; (b) it has the disadvantage that it only considers very limited (transaction) characteristics, and does not take into consideration other important database characteristics such as the size of instance variables; (For example, when we change the size of the instance variable name in the PROJECT class from 40 to 10 bytes, the cost-driven approach would respond to the change and produce a new optimal partitioning scheme. But as the affinity-based approach does not consider the sizes of the instance variables, it does not adapt to the change and would retain the old partitioning scheme.) and (c) it is more suitable for determining an initial vertical class partitioning scheme.

(3) **HCHA**: (a) it has a comparable predicting power as CDA; (b) it uses the cost-based formulae to compare for optimality, thus can respond to database characteristic changes (such as the change in instance variable sizes); (c) it avoids the exhaustive enumeration of all the possible partitioning schemes, hence the computation cost is acceptable for a large number of instance variables; (d) as with other hill-climbing algorithms, there is a chance of not finding the globally optimal solution; and (e) it is most suitable both for determining an initial vertical class partitioning scheme and for a production OODB system.
4.5 Summary

Due to the encapsulation property of OODBs, methods govern the nature of optimal (or near optimal) vertical partitioning schemes. Thus in this chapter, we take into consideration the methods being invoked during OODB processing to generate optimal or near-optimal vertical partitioning schemes. In our earlier work [29,30], we had already shown the utility and effectiveness of vertical partitioning in OODB systems. In this chapter, we have developed, based on MDGs (method dependency graphs), a general purpose cost model for method execution, and applied this model to come up with vertical partitioning algorithms. Two algorithms have been developed to exploit method execution for deriving optimal or near-optimal vertical partitioning schemes. The first algorithm is the cost-driven algorithm which uses the cost model for method execution to exhaustively search all the partitioning schemes so as to find the optimal vertical partitioning scheme. This algorithm is useful for comparing the effectiveness of the other algorithms, but may not be practical since it has a high cost of $O(n^n)$ where $n$ is the number of instance variables. The second algorithm, HCHA, first applies an affinity-based algorithm [64] to get an initial partitioning scheme, then applies hill-climbing heuristic to improve this solution by using the cost model for method execution. It is shown that the HCHA (which is significantly more efficient than the cost-based approach) generates mostly optimal or near optimal schemes. The HCHA approach thus represents a good compromise and it will generate a solution that is at least as good as the solution provided by the affinity-based approach, and in most cases, can yield the optimal partitioning scheme. But as with all hill-climbing algorithms, there are chances that the algorithm cannot generate the optimal solution and be trapped in a local minima.

We further note that as data in an OODB is accessed through methods. With the help of method transformation, our method-induced VCP technique can easily support fragmentation transparency [50,49]. This fragmentation transparency support places our method-induced VCP technique in a better position to be incorporated into an OODBMS than the affinity-based approach.

In order to obtain effective VCP scheme using our method-induced VCP technique, we need to obtain high quality estimations of quite a number of method execution parameters. This is an obstacle if the underlying OODBMS cannot provide accurate estimations on the various method execution parameters.
Chapter 5

Structural Join Index Hierarchy Framework

An OODB model facilitates modeling database requirements of next generation database (NGDB) applications through complex objects. For NGDB applications, complex object retrieval has a major impact on the cost of processing the queries. Without any index or access support structures, complex object retrieval is processed by pointer chasing. But the cost of query processing using pointer traversal is very expensive, especially when: (1) the objects are large; (2) component objects to be retrieved are deep inside the class composition hierarchy; (3) traversing in the reverse direction of the path expression is required (due to the absence of reverse pointers); and (4) structural information (the links between the objects) need to be retrieved.

To overcome the problems of expensive pointer traversal, many techniques have been proposed to expedite the query processing. These include indexing [7,51,76], function materialization [46], complex object assembly [43], and clustering [26]. In this chapter, we develop the concepts, and formulate a framework of Structural Join Index Hierarchy (SJIH) that encompasses the existing OODB indexing methods including Nested index, Path index [7], Access Support Relation [51] and Join Index Hierarchy [76]. A cost model for processing the queries and maintaining the indices is developed for evaluating the SJIHs, as well as for comparing different complex object retrieval indexing schemes. Our results show that SJIH indexing mechanisms not only facilitate efficient retrieval of complex objects, but also outperform many of the existing indexing mechanisms for complex object retrieval.

The rest of the chapter is organized as follows: Section 5.1 stipulates the development of our OODB indexing framework, namely structural join index hierarchy (SJIH). Section 5.2 presents an analytical cost model. Section 5.3 provides performance evaluation of our SJIH frame-
work. Section 5.4 presents the pros and cons of using different SJIHS and guidelines for index designers and finally, Section 5.5 presents the summary.

5.1 SJIH Mechanisms

5.1.1 Basic definitions

As mentioned earlier, the aim of the thesis is to formulate a framework of OODB indexing that facilitates efficient complex object retrieval. We use the Join index approach to store OIDs in the form of tuples for efficient complex object retrieval. In this OODB indexing framework, we present the following index categories: (1) base join index (BJI) ([76]), (2) derived join index (DJI) ([76]), (3) hyper join index (HJI) (our contribution), and (4) structural join index (SJI) (our contribution). We first adopt from [7] some basic definitions that are needed for our subsequent discussion.

**Definition 2**

Given a class composition hierarchy (CCH), a path is defined as \( c_0.A_1.A_2.\ldots.A_n \) \( (n \geq 1) \)

where: (a) \( c_0 \) is the root class in the CCH; (b) \( A_i \) is an attribute of class \( c_0 \); (c) \( A_i \) is an attribute of the class \( c_i \) in the CCH, such that \( c_i \) is the domain of attribute \( A_{i-1} \) of class \( c_{i-1}, 1 < i \leq n \).

**Definition 3**

Given a path \( p = c_0.A_1.A_2.\ldots.A_n \), a complete instantiation of \( p \) is defined as a sequence of \( n + 1 \) objects, denoted as \( o_0.o_1.o_2.\ldots.o_n \) where: (a) \( o_0 \) is an instance of \( c_0 \); (b) \( o_i \) is the value of attribute \( A_i \) of object \( o_0 \); (c) \( o_i \) is the value of attribute \( A_{i-1} \) of object \( o_{i-1} \) (i.e., \( o_{i-1}.A_{i-1} = o_i \) if \( A_{i-1} \) is single-valued; \( o_i \in o_{i-1}.A_{i-1} \) otherwise), \( 1 < i \leq n \).

**Definition 4**

Given a path \( p = c_0.A_1.A_2.\ldots.A_n \), a partial instantiation of \( p \) is defined as a sequence of objects \( o_a.\ldots.o_b, 0 \leq a < b \leq n \) where: (a) \( o_a \) is an instance of a class \( c_a \) in the CCH; (b) \( o_i \) is the value of attribute \( A_{i-1} \) of object \( o_{i-1}, a < i \leq b \).

BJI and DJI are defined over a single path, whereas HJI and SJI are defined over multiple paths. In the following section we describe the specification of an OODB schema with multiple paths to facilitate the discussion on the definitions of HJI and SJI.

5.1.2 Representation of OODB schema with multiple paths

For a class composition hierarchy as shown in Figure 40 with \( Q \) paths \( (Q > 1) \), we have the follow-
ing notation: (a) $C_0$ is the root class; (b) path 1 has a total of $n_1$ classes (excluding the root class), path 2 has a total of $n_2$ classes, ..., and path $Q$ has a total of $n_Q$ classes.

![Class Composition Hierarchy](image)

**Figure 40: Class Composition Hierarchy**

5.1.3 Definitions of the different indices in SJIH

**Base join index (BJI)**

**Definition 5**

Given a path $P = C_0.A^p_1.A^p_2....A^p_{n_p}$ (i.e., the $p$ th path in the CCH with $1 \leq p \leq Q$), a BJI is defined as $BJI = \{(\rho^p_i, \rho^p_{i+1})\}$ where $\rho^p_i, \rho^p_{i+1}$ is a partial instantiation of path $P$ with $0 \leq i < n_p$. That means BJI is a binary relation with the two attributes being the OIDs of two neighbouring classes.

Our subsequent discussion is also based on the running example schema (in Section 2.6) adapted from the OO7 Benchmark [14]. The schema and occurrence are repeated here (Figures 41 and 42) to facilitate our discussion.
The most primitive index under SJIH is the BJI: a BJI is used to support navigation between two adjacent classes. Furthermore, BJIs are also useful in building up more complex higher level indices. Figure 43 shows an example BJI between the classes AtomicPart and CompositePart. Note that BJIs can facilitate the query processing in both the forward and reverse direction.
Derived join index (DJI)

Definition 6
Given a path \( P = A_{p_1} A_{p_2} \ldots A_{p_n} \), a DJI is defined as

\[
\text{DJI} = \{(d_i, d_j, \text{dup}) \}
\]

where \( d_i, d_j \) is a partial instantiation of path \( P \) with \( 0 \leq i < j \leq n \). A DJI is a ternary relation with the first two attributes being the OIDs of two classes along the same path, and the third attribute (\( \text{dup} \)) is a duplicating factor which records the number of duplicates. As in [76], a \( \text{dup} \) value greater than 1 means that there is more than one connection relating the OIDs of two classes in the OODB. These duplicate factors facilitate the management of the impact of object deletions on the DJI.

DJIs are “derived” from the BJIs. While a BJI relates two adjacent classes in the OODB schema, a DJI relates classes that are one or more classes apart and on the same path. In Figure 44, the DJI (between classes \( \text{Connection} \) and \( \text{CompositePart} \)) is derived from the two BJIs: one between classes \( \text{Connection} \) and \( \text{AtomicPart} \) and the other between classes \( \text{AtomicPart} \) and \( \text{CompositePart} \). In a more complex OODB schema, a DJI can also be derived from other DJIs and/or BJIs. Similar to BJI, we note that DJI also facilitates the query processing in both the forward and reverse direction.

Hyper join index (HJI)

Definition 7
Given two distinct paths \( P_1 = A_{p_1} A_{p_2} \ldots A_{p_{n_1}} \) and \( P_2 = A_{p_1} A_{p_2} \ldots A_{p_{n_2}} \) with
$1 \leq p_1, p_2 \leq Q$, HJI is defined as

$$\text{HJI} = \{ (o^p_{i_1}, o^p_{j_1}, \text{dup} ) \}$$

where $o^p_{i_1}$ and $o^p_{j_1}$ are partial instantiations of paths $p_1$ and $p_2$, respectively, with $0 \leq i_1 \leq np_1$, $0 \leq j_1 \leq np_2$. Like a DJI, a HJI is also a ternary relation, where the first two attributes are the OIDs of two classes (over two paths). The third attribute is still the duplicating factor.

HJIs also relate objects of two classes in an OODB schema and consists of two OIDs. But HJI can involve classes from two different paths. Figure 45 shows an example HJI between the classes Documentation and Engineer. Note that a DJI is a special case of a HJI when the two classes are from the same path. Similar to DJIs and BJIs, HJIs also facilitate the query processing in both directions. For example, in Figure 45 we can traverse in both directions: one direction is from class Documentation to class Engineer; the other direction is from class Engineer to class Documentation.

**Structural join index (SJI)**

**Definition 8**

Given $g$ distinct paths $P_a = C_0.A^{p_a}_{1}.A^{p_a}_{2}...A^{p_a}_{np_a}$, $P_b = C_0.A^{p_b}_{1}.A^{p_b}_{2}...A^{p_b}_{np_b}$, $...$, and $P_g = C_0.A^{p_g}_{1}.A^{p_g}_{2}...A^{p_g}_{np_g}$ (with $g \geq 1$ and $1 \leq p_a, p_b, ..., p_g \leq Q$), a SJI is defined as

$$\text{SJI} = \{ (o^{p_a}_{i_1}, ..., o^{p_a}_{i_p}, o^{p_b}_{j_1}, ..., o^{p_b}_{j_p}, ..., o^{p_k}_{k_1}, ..., o^{p_k}_{k_p}, \text{dup} ) \}$$

where $o^{p_a}_{i_1}, ..., o^{p_a}_{i_p}$, $o^{p_b}_{j_1}, ..., o^{p_b}_{j_p}$, ..., and $o^{p_k}_{k_1}, ..., o^{p_k}_{k_p}$ are partial instantiations of paths $P_a$, $P_b$, ..., and $P_g$, respectively, with $0 \leq i_1 < ... < i_p \leq np_a$, $0 \leq j_1 < ... < j_p \leq np_b$, ..., and $0 \leq k_1 < ... < k_p \leq np_g$. If the SJI covers a total of $h$ OIDs, then the SJI is a $h + 1$-ary relation, where the first $h$ attributes are the OIDs of the $h$ classes (over the $g$ paths). The last attribute is still the duplicating factor.

An SJI is a sophisticated structure as it may contain more than two OIDs from classes in an OODB schema. In fact, an SJI may even cover the whole schema structure (complete SJI), though more often it covers part of the schema structure (partial SJI). From the design point of view, an SJI is a high level abstraction of the original OODB schema that contains the OID information about the classes that are covered by the SJI. As shown in Figure 46, the SJI covers classes Design, Engineer, Office, Documentation and AtomicPart. Note that SJI facilitates the query processing in multiple directions by building non-clustered indices.
5.1.4 Relationships among different indexing methods

Concerning the applicability of the different indices, we note that for applications where Multi index, Join index, Nested index, Path index and Access Support Relation are useful, they will also imply BJI, DJI, HJI and SJI are equally useful. This is because BJI, DJI, HJI, and SJI are actually their extensions. Furthermore, HJI and SJI can also be used by applications that need to access multiple paths. Hence our join indices will have even better performance in such applications when compared with other indexing methods. From the above discussion, the various indices can be related in the following ways as shown in Figure 47:

- BJI and Multi Index (MI) both index on two neighbouring classes, but BJI is more general and can support both forward and reverse traversals.
- DJI and Nested Index (NI) both index on two classes along a path. But DJI supports traversal in both directions. HJI is a multi-path extension to DJI.
- ASR and Path index (PI) cover all the classes along a path, but ASR facilitates traversal in both directions. Further, SJI is a multi-path extension to ASR and PI.
- SJIH is a multi-path extension to JIH.

Within our framework of SJIH, the different categories of join indices can be related in the
following ways:

- when the two classes are adjacent to each other, BJI is a special case of DJI;
- when the two classes are on the same path, DJI is a special case of HJI;
- when the index involves only two classes, HJI is a special case of SJI.

Figure 47: Relationships among the different indexing methods

5.1.5 Types of Structural Join Index Hierarchy

In a query processing environment with multiple queries, one single indexing method is often not enough to expedite all queries. Furthermore, the user requirements and query patterns may change, requiring accesses to different classes during query processing. Our framework solves this problem by having a Structural Join Index Hierarchy (SJII). An SJII can consist of a set of partial SJIIs (or even a complete SJI) coupled with some simpler BJIIs, DJIIIs and HJIIs. An ideal SJII should expedite most of the queries in the query processing environment. When the query pattern changes, we can use the existing indices in the SJII to form new indices. The following classifications of SJIIIs are introduced for a given OODB schema:

- **Complete-SJII**: It is a single large SJI that covers every class in the schema. Complete-SJII is the most powerful as all the OID information and hence all the relationships between objects in all the classes is captured. The more the number of classes an SJII has, the more complex it is. The major problem with Complete-SJII is its large storage and update overhead, especially for a high degree of sharing between composite objects and their component objects.

- **Partial-SJII**: A Partial-SJII may contain a number of partial SJIIs and some other simpler BJIIIs, DJIIIs and HJIIs. Since not all the OID information is stored but is spread over a number of indices, we may need to combine these indices to obtain the results for query execution. Partial-SJII has the advantage of lower update cost, but it may imply higher query processing cost. For cases with a high degree of sharing, the storage overhead will be much lower than that of the Complete-SJII.
• **Base-SJIH**: It is the collection of *all* the BJI s between every two neighbouring classes in the OODB schema. Query processing requires combining the relevant BJI s to obtain the result. The main advantage of the Base-SJIH is its low storage overhead even in the case of a high degree of sharing. Another advantage is that it is efficient to update a BJI. The main problem with the Base-SJIH is that the performance of query processing may not be as good as that of the Complete-SJIH or a Partial-SJIH, due to the large number of BJI s involved.

### 5.2 Query Processing and Cost Model

#### 5.2.1 Query processing strategy
In Table 10, we consider some examples of complex object retrieval queries which are in OQL (*Object Query Language*) notation, and provide further discussion on the strength/weakness of each individual indexing method by means of the example queries. Furthermore, the query processing strategies of each indexing method are also discussed. In our discussion, we classify queries into two categories:

- in a *forward query*, we start from the root class objects that satisfy a predicate, and traverse down the path to find the requested objects.
- in a *backward query*, we start from objects that belong to the ending class of a path and traverse up the path in the reverse direction.

<table>
<thead>
<tr>
<th>Queries</th>
<th>Discussion</th>
</tr>
</thead>
</table>
| `select OID(C), OID(C.atomic-part) from C in CompositePart where C.atomic-part.name="Resistor"` | • This is a query on two adjacent classes along a single path.  
• This is a uni-directional query -- backward query from class *AtomicPart* back to *CompositePart* (with the predicate on the *AtomicPart* class). We can make use of the MI. If there are other queries that also require searching in the forward direction, then we can make use of the BJI. |

![Figure 48: Query index 1](image)
select OID(D), OID(D.composite-part.atomic-part) from D in Design
where D.composite-part.atomic-part.name="Resistor"

This is a query on two non-adjacent classes along a single path.
This is a backward query. We can make use of the NI. If forward traversal is also required, then we can make use of the DJI.

select OID(D), OID(D.composite-part),
OID(D.composite-part.atomic-part) from D in Design
where D.composite-part.atomic-part.name="Resistor"

This is a query on three contiguous classes along a single path.
We can make use of the ASR.
This is a backward query. If forward traversal is also required, we can still make use of the ASR.

select OID(D.team.engineer),
OID(D.composite-part.documentation) from D in Design
where D.composite-part.documentation.title="IC"

This is a query on two classes along two different paths.
If we use the conventional OODB indexing, we need 4 MIs, 2 NIs, or 2 ASRs to support this query. These are inefficient because:
- MI requires quite a number of MIs (4 in this case) to support query processing.
- NI requires 2 NIs to support the query; and the 2 NIs will involve the root class Design, although it is not requested by the query.
- ASR requires 2 ASRs to support the query; and the 2 ASRs will involve the Design, Team and CompositePart classes, although they are not requested by the query.
- The direction of the query is from class Documentation towards Engineer. We can make use of the HJI. If there are bi-directional query processing requirements, then we can still make use of the HJI.
We now discuss the query processing strategy using our SJIH. If there is only one query, construct an SJIH (i.e., a set of SJI(s)) that covers all the relevant OIDs of the classes accessed by the query. The predicate in the \texttt{where} clause is evaluated first. We illustrate the query processing strategy for the following forward query (in OQL notation and as shown in Figure 53):

\begin{verbatim}
select OID(D), OID(D.team.engineer), OID(D.composite-part.atomic-part),
       OID(D.composite-part.atomic-part.connection)
from D in Design
where D.name = "Motherboard"
\end{verbatim}

This is a query on five classes along four different paths.
If we use the conventional OODB indexing, we need 6 MIs, 4 NIs, or 4 ASRs to support this query. These are inefficient since:
- MI requires a large number of MIs (6 in this case) to support query processing.
- NI requires 4 NIs to support the query; and the 4 NIs will all contain the OIDs of the root class \textit{Design}, leading to redundancy.
- ASR requires 4 ASRs to support the query; and they will involve the \texttt{Team} and \texttt{CompositePart} classes, although they are not requested by the query.
This is a forward query, the direction is from the root class \textit{Design} towards the classes at or near to the leaf-level of the schema. We can make use of the SJI. If there are other multi-directional query processing requirements, then we can still make use of the SJI.
To facilitate the presentation, we assume that an SJIH = \{ SJ1, SJ2, SJ3 \} (as shown in Figure 54) is available for query processing, where:

- **SJ1** is a SJI involving classes Design, Documentation and AtomicPart,
- **SJ2** is a DJI involving classes Design and Engineer,
- **SJ3** is a BJI involving classes AtomicPart and Connection.

First, we obtain those Design class OIDs that have "Motherboard" as their name. After that, (a) the Design class OIDs are used to access the SJ1 to obtain the OIDs of class AtomicPart; (b) these resulting OIDs are used to access SJ3 (using the OIDs of class AtomicPart) to obtain the OIDs of class Connection; (c) Design class OIDs (in (a) above) are used to access SJ2 to obtain the OIDs of class Engineer. Finally, we collect all the resultant OIDs (in (a), (b), and (c)) and return them as a collection of OID tuples. Similar strategies can also be applied to process backward queries.

For a query processing environment involving a set of queries, we can also make use of the above strategy. But now it is more difficult to find an optimal SJIH that covers all the OIDs of the classes in all the select lists of the queries. Though the Complete-SJIH and Base-SJIH may be two possible candidates, they may not be the optimal choice. The index selection problem of finding the optimal SJIH for processing a given set of queries is a hard problem. In Chapter 6, we formulate this optimization problem with the help of the cost model and design a heuristic algorithm.

### 5.2.2 Cost Model Parameters

A cost model for our SJIH framework is essential for comparing the effectiveness of SJIH with different join indices. Table 11 presents the basic cost model parameters, while Table 12 specifies some of the parameter values used which are taken from [74,7,51,76,37,30,6].
Table 11: Cost model parameters for SJII

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$|C^p|_i$</td>
<td>cardinality of class collection $C^p_i$ (i.e., the number of objects in the $i$-th class along the class composition hierarchy along the $p$-th path). For single path, we denote it as $</td>
</tr>
<tr>
<td>$</td>
<td>C^p_i</td>
</tr>
<tr>
<td>$f^p_i$</td>
<td>forward fan-out -- average number of references from an object in $C^p_i$ to objects in $C^p_{i+1}$ for the $p$-th path</td>
</tr>
<tr>
<td>$r^p_{i+1}$</td>
<td>reverse fan-out -- average number of objects in $C^p_{i+1}$ referencing the same object in $C^p_i$</td>
</tr>
<tr>
<td>$SC^p_i$</td>
<td>size of object (in byte) of class $C^p_i$</td>
</tr>
<tr>
<td>$SOID$</td>
<td>number of bytes for storing an object identifier</td>
</tr>
<tr>
<td>$SM$</td>
<td>number of bytes for storing the duplicate factor</td>
</tr>
<tr>
<td>$STJI$</td>
<td>number of bytes for a tuple in a Join index</td>
</tr>
<tr>
<td>$PS$</td>
<td>page size of the file system</td>
</tr>
<tr>
<td>$POF$</td>
<td>average page occupancy factor in the $B^+$ tree</td>
</tr>
<tr>
<td>$BTF$</td>
<td>fan-out of the $B^+$ tree</td>
</tr>
<tr>
<td>$fwd(i, j, k, p)$</td>
<td>average number of distinct objects in $C^p_j$ referenced by a set of $k$ objects in $C^p_i$ for any $i &lt; j$ for the $p$-th path. For single path, we denote it as $fwd(i, j, k)$.</td>
</tr>
<tr>
<td>$bwd(i, j, k, p)$</td>
<td>average number of distinct objects in $C^p_i$ referencing a set of $k$ objects in $C^p_j$ for any $i &lt; j$ for the $p$-th path. For single path, we denote it as $bwd(i, j, k)$.</td>
</tr>
<tr>
<td>$n$</td>
<td>number of tuples in an index</td>
</tr>
<tr>
<td>$m$</td>
<td>number of pages occupied by an index</td>
</tr>
<tr>
<td>$k$</td>
<td>number of tuples retrieved from an index during query processing</td>
</tr>
<tr>
<td>$Sel$</td>
<td>selectivity of the predicate in the OQL query</td>
</tr>
<tr>
<td>$ref^p_i$</td>
<td>number of object references in class $C^p_i$ for the $p$-th path</td>
</tr>
</tbody>
</table>

Table 12: Cost model parameter values used

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SOID$</td>
<td>8</td>
</tr>
<tr>
<td>$SM$</td>
<td>4</td>
</tr>
<tr>
<td>$PS$</td>
<td>8192</td>
</tr>
<tr>
<td>$POF$</td>
<td>70%</td>
</tr>
<tr>
<td>$BTF$</td>
<td>409</td>
</tr>
</tbody>
</table>

5.2.3 Storage cost

We store the OIDs of an SJI in the form of tuples in the leaf levels of a clustered $B^+$ tree index. We can also build non-clustered indices on the SJI tuples if required. The index tuple consists of the
OIDs of the classes on which the SJI is defined. The tuples are stored as a relation with no redundancy. If there are duplicate SJI tuples, a "duplicating factor" is used to record the total number of duplicates. This duplicating factor can facilitate the management of the index entries.

![Diagram](image.png)

Figure 55: SJI with clustered and non-clustered indices

To conduct our following discussion, we assume a generic SJI as shown in Figure 55 which occupies $m$ pages of storage and with $n$ tuples. This SJI has a clustered index on OIDs of class $C_1$; furthermore, it has non-clustered indices on OIDs of the classes $C_2$, ..., $C_X$. The storage cost of the SJI is

$$\text{Storage Cost} = \text{Storage Cost for Clustered Index} + \text{Storage Cost for Non-Clustered Indices}.$$  

As in [76], we concentrate on the storage cost of the leaf-level nodes of the clustered index that stores the SJI tuples. We neglect the storage cost for the non leaf-levels nodes of the clustered index as it is small when compared with the leaf-level. Furthermore, as not every class in an SJI has a non-clustered index on it, for uniform comparison between different SJJIs, the storage costs for non-clustered indices are also neglected.

The storage cost of the SJI is given by: $m = \left\lceil \frac{n \times STJI}{PS \times POF} \right\rceil$, where $n$ is the number of SJI tuples, $STJI$ is the length of the SJI tuples, $PS$ is the page size of the system, and $POF$ is the page occupancy factor of the $B^+$ tree index. In the above formula, $PS$ and $POF$ are constants for a given OODB system, the variables $STJI$ and $n$ are described below.

**Length of SJI tuples**

For DJI, HJI, and SJI which can have the duplicating factors in their index tuples, $STJI = \text{numOID} \times \text{SOID} + \text{SM}$, where $\text{numOID}$ is the number of OIDs in the index (i.e., the number of classes involved), $\text{SOID}$ and $\text{SM}$ are the number of bytes used to store an OID and the duplicating factor, respectively. But for a BJI, $STJI = 2 \times \text{SOID}$ because a BJI does not have the duplicating fac-
tor. The storage cost is directly proportional to $STJI$, which means as the number of OIDs in an SJI tuple increases, the storage cost also increases.

**Number of SJI tuples**

For a forward query, we start with the root class $C_0$. The number of SJI tuples is given by:

$$n = |C_0| \times MultiplyingFactor.$$  

The $MultiplyingFactor$ is highly dependent on the forward fan-out values of the different paths in the OODB schema. For a backward query, we start with one of the ending class $C_{p_n}$ (in the $p$th path of the OODB schema). The number of SJI tuples is given by:

$$n = |C_{p_n}| \times MultiplyingFactor.$$  

The details for calculating $MultiplyingFactor$ are given below.

**Formula for calculating the $MultiplyingFactor$**

Let us first review the semantics of "composite reference" in a class composition hierarchy (CCH). In [42], an object may recursively reference a number of other objects. An object $O'$ has a reference to another object $O$, if $O'$ contains the OID of $O$. We distinguish two types of reference from one object to another: weak and composite. A composite reference in turn may be of two types: exclusive and shared. A weak reference is the standard reference in object oriented systems and carries no special semantics. A composite reference is a weak reference augmented with the IS-PART-OF relationship; a composite reference from $O'$ to $O$ means that $O$ is a part of $O'$. The semantics of a composite reference are further refined on the basis of whether an object is a part of only one object or more than one object. An exclusive composite reference is further refined on the basis of whether an object is a part of only one object or more than one object. An exclusive composite reference from $O'$ to $O$ means that $O$ is a part of only $O'$; while a shared composite reference from $O'$ to $O$ means that $O$ is a part of $O'$ and possibly other objects.

[42] further refines the semantics of a composite reference, either exclusive or shared, on the basis of whether the existence of an object depends on the existence of its parent object; that is, a composite reference may be dependent or independent. An existence dependent composite reference from $O'$ to $O$ means that the existence of $O$ depends on the existence of $O'$; while an existence independent composite reference does not carry this additional semantics. The deletion of an object will trigger recursive deletion of all objects referenced by the object through existence dependent composite references (both exclusive and shared).

Exclusive and shared composite references will have great influence on the number of SJI tuples in the CCH with multiple paths. But existence dependent and independent composite references only affect the cascade deletions of the object instances when the ancestor object is deleted. These existence dependent and independent composite references will not have any effect on the number of SJI tuples in the CCH.
We find that the *constrained pair-up*\(^1\) between the classes in the different branches of the CCH has great influence on the number of SJI tuples. Let us illustrate this by the example schema as shown in Figure 56. We will discuss 4 different cases:

- **Case (A)**: classes *Team* and *CompositePart* are constrained pair-up (as shown in Figure 57), the dotted lines show the constrained pair-up between the two classes. Furthermore, the *Design-Team* and *Design-CompositePart* composite references are *exclusive*.

- **Case (B)**: classes *Team* and *CompositePart* are constrained pair-up (as shown in Figure 58). But the *Design-Team* and *Design-CompositePart* composite references are *shared*.

- **Case (C)**: classes *Team* and *CompositePart* are unconstrained pair-up (as shown in Figure 59), the objects in class *Team* and *CompositePart* can be paired up freely. Furthermore, the *Design-Team* and *Design-CompositePart* composite references are *exclusive*.

- **Case (D)**: classes *Team* and *CompositePart* are unconstrained pair-up (as shown in Figure 60). But the *Design-Team* and *Design-CompositePart* composite references are *shared*.

| Case (A) | The total number of SJI tuples is 5. |
|-----------------|-----------------|----------------|
| OID Design | OID Team | OID CompositePart |

| Case (B) | The total number of SJI tuples is 10. |
|-----------------|-----------------|----------------|
| OID Design | OID Team | OID CompositePart |

---

1. By *constrained pair-up* (between two classes A and B), we mean that there will be constraints on the pair up of the actual object (between classes A and B). If the two classes A and B are *unconstrained pair-up*, the objects in the two classes can be paired up freely (without any constraint).
Case (C)
The total number of SJI tuples is 13.

<table>
<thead>
<tr>
<th>OID Design</th>
<th>OID Team</th>
<th>OID CompositePart</th>
</tr>
</thead>
</table>

Case (D)
The total number of SJI tuples is 24.

<table>
<thead>
<tr>
<th>OID Design</th>
<th>OID Team</th>
<th>OID CompositePart</th>
</tr>
</thead>
</table>

---

Figure 56: OODB Schema for SJH

Figure 57: Constrained pair-up between classes Team and CompositePart, the Design-Team and Design-CompositePart composite references are exclusive
As seen from the Figures 57 to 60 and Tables 13 and 14, we have the following conclusions regarding the number of SJI tuples in different cases. Before the detailed discussion, we term the forward fan-out from class Design to class Team and class Design to class CompositePart as f1 and f2, respectively.

In Table 14, the fifth column predicts the number of SJI tuples using the max formula -- the cardinality of class Design times the maximum between f1 and f2. The sixth column predicts the number of SJI tuples using the product formula -- the cardinality of class Design times the product of f1 and f2. Note that the predicted number of SJI tuples using the max formula is much smaller than the predicted number using the product formula. The general observations (as highlighted in Table 14) are:

- if the Team and CompositePart classes are constrained pair-up and if the composite references from class Design to class Team or class CompositePart are exclusive, the number of SJI tuples is correctly predicted by the max formula.
- if the Team and CompositePart classes are constrained pair-up and if the composite references from class Design to class Team or class CompositePart are shared, the number of SJI tuples is
close to (but not exactly) the number predicted by the max formula. When we calculate the predicted number using the max formula, we use f2, which is equal to 4. We observe that the deviation of the actual number of SJI tuples with respect to the predicted number using max formula: 10/8 = 1.25 and is less than f1 (which is 3), i.e., the deviation is bounded above by the forward fan-out factor of the other branch (not used in the calculation of the max formula). We further note that there is sharing in both branches.

- if the Team and CompositePart classes are unconstrained pair-up, the number of SJI tuples is correctly predicted by the product formula. (This is regardless of whether the composite references from class Design to class Team or class CompositePart are exclusive or shared).

<table>
<thead>
<tr>
<th>Case</th>
<th>Constrained Pair-up between classes Team &amp; CompositePart</th>
<th>Exclusive/Shared Composite Reference (from Design class)</th>
<th>Cardinality of Design class</th>
<th>f1</th>
<th>f2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>constrained</td>
<td>exclusive</td>
<td>2</td>
<td>2.5</td>
<td>2.5</td>
</tr>
<tr>
<td>B</td>
<td>constrained</td>
<td>shared</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>unconstrained</td>
<td>exclusive</td>
<td>2</td>
<td>2.5</td>
<td>2.5</td>
</tr>
<tr>
<td>D</td>
<td>unconstrained</td>
<td>shared</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case</th>
<th>Constrained Pair-up between classes Team &amp; CompositePart</th>
<th>Exclusive/Shared Composite Reference (from Design class)</th>
<th>Actual number of SJI tuples</th>
<th>Predicted number using max formula</th>
<th>Predicted number using product formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>constrained</td>
<td>exclusive</td>
<td>5</td>
<td>5</td>
<td>12.5</td>
</tr>
<tr>
<td>B</td>
<td>constrained</td>
<td>shared</td>
<td>10</td>
<td>8</td>
<td>24</td>
</tr>
<tr>
<td>C</td>
<td>unconstrained</td>
<td>exclusive</td>
<td>13</td>
<td>5</td>
<td>12.5</td>
</tr>
<tr>
<td>D</td>
<td>unconstrained</td>
<td>shared</td>
<td>24</td>
<td>8</td>
<td>24</td>
</tr>
</tbody>
</table>

The intuition from the above observations are:

- if there is constrained pair-up between the classes in the different branches of the CCH, the number of SJI tuples will not grow very fast as the forward fan-out increases. Furthermore, it can be predicted exactly (after applying the ceiling function) in the exclusive case and approximately in the shared case by multiplying the cardinality of the root class with the maximum of the forward fan-outs in the different branches.

- if there is no constraint for the pair-up between the classes in the different branches of the CCH, the number of SJI tuples will grow very fast as the forward fan-out increases. Furthermore, it can be predicted exactly (after applying the ceiling function) in both exclusive case and shared case by multiplying the cardinality of the root class with all the forward fan-outs in the different branches.
In cases where there is constrained pair-up between the classes in the different branches, as the number of SJI tuples is not high, the storage overhead for the use of SJI will not be high. Furthermore, the index retrieval cost, navigation cost and index maintenance cost will also be low. This implies that the use of SJI is very effective.

In a semantically rich OODB schema, the constrained pair-up (both exclusive and shared) case composite references will be a more realistic representation for the real world application. Hence in the later sections, we concentrate our discussion/experimentation on the constrained pair-up case.

To generalize the above discussion, we first define the **MultiplyingFactor (MF)**:

**Definition 9**

For a subtree (rooted by class \( C_i \)) in an OODB schema, the total number of SJI tuples in that particular subtree is given by \( n = \|C_i\| \times MF(C_i) \). This means that the total number of SJI tuples is given by multiplying the MultiplyingFactor \( MF(C_i) \) with the cardinality of the root class of the subtree.

(I) **Subtree with only one level**

This is the trivial situation, and class \( C_i \) is a leaf node in the OODB schema, we let \( MF(C_i) = 1 \).

(II) **Subtree with two levels**

Another situation is when class \( C_i \) has \( k \) children (classes \( C_{j_1}, C_{j_2}, \ldots, C_{j_k} \)), and all the \( k \) children do not have any descendants. This means that the subtree rooted by class \( C_i \) is a tree with two levels. The \( MF \) is dependent on the constrained/unconstrained pair-up between classes in the different paths. There are three cases:

- **Unconstrained pair-up**: when all the classes in the OODB schema are unconstrained pair-up, then \( MF \) is the product of all the forward fan-outs in all the paths:

  \[
  MF(C_i) = \prod_{j=j_1}^{j_k} f_{C_i - C_j} \text{, where } f_{C_i - C_j} \text{ is the forward fan-out from class } C_i \text{ to its child class } C_j. 
  \]

- **Constrained pair-up and exclusive**: when all the classes are constrained pair-up and if all relationships are exclusive, then \( MF \) is given by the maximum of the product of forward fan-outs of each individual path:

  \[
  MF(C_i) = \max( f_{C_i - C_{j_1}} \cdots , f_{C_i - C_{j_k}} ).
  \]

- **Constrained pair-up and shared**: similar to the above but if some relationships are shared, then \( MF \) is approximated by a constant \( K \) (with \( K > 1 \)) times the maximum of the product of
each individual path:

\[ MF(C_i) = K \times \max(f_{c_{i1}, \ldots, c_{i\alpha}}) \]

The constant \( K \) depends on the degree of sharing/forward fan-out between the class \( C_i \) and its *shared* child classes. Further, \( K \) is bounded above by \( K \leq \prod_j f_{c_i - c_j} \), where \( j \) in the product ranges over: (a) all shared child classes; and (b) these shared child classes should not be from the path with maximum forward fan-out (as it is already accounted for in the calculation of \( \max(f_{c_{i1}, \ldots, c_{i\alpha}}) \)). Note that as we only consider shared child classes (in the calculation of \( K \)), in cases where there is only one or a few shared child classes, \( K \) is close to 1, i.e., the \( MF \) is close to that of the previous constrained pair-up and exclusive case. But when there are a large number of shared child classes, the \( MF \) is close to that of the first unconstrained pair-up case.

In general, the \( MF \) is calculated by:

\[ MF(C_i) = K \times OP(f_{c_i - c_j}) \]

, where \( K \) and \( OP \) are summarized in Table 15.

**Table 15: \( K \) and \( OP \)**

<table>
<thead>
<tr>
<th>Case</th>
<th>Constrained Pair-up between classes Team &amp; CompositePart</th>
<th>Exclusive / Shared Composite Reference (from Design class)</th>
<th>( K )</th>
<th>( OP ) (Operation on the forward fan-out values)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>constrained</td>
<td>exclusive</td>
<td>1</td>
<td>Maximum</td>
</tr>
<tr>
<td>B</td>
<td>constrained</td>
<td>shared</td>
<td>( 1 &lt; K \leq \prod_j f_{c_i - c_j} )</td>
<td>Maximum</td>
</tr>
<tr>
<td>C</td>
<td>unconstrained</td>
<td>exclusive</td>
<td>1</td>
<td>Product</td>
</tr>
<tr>
<td>D</td>
<td>unconstrained</td>
<td>shared</td>
<td>1</td>
<td>Product</td>
</tr>
</tbody>
</table>

(III) Subtree with any levels

The most general situation is when class \( C_i \) has \( k \) children, and some/all of the \( k \) children may have their own descendants. This means that the subtree rooted by class \( C_i \) can be of any levels. As in the previous situation, \( MF \) is dependent on the constrained/unconstrained pair-up between classes in the different paths. We have the following recursive definition for \( MF \):

\[ MF(C_i) = K_i \times OP_i (f_{c_i - c_j} \times MF(C_j)) \]

This means that the \( MF \) of a subtree rooted by class \( C_i \) is recursively defined upon the \( MFs \) of its child classes \( C_j \). The \( K_i \) and \( OP_i \) are determined by the nature of the composite reference, i.e., constrained/unconstrained and exclusive/shared. Refer to Table 15 for the \( K_i \) and \( OP_i \) in the different cases.

Let us illustrate the above discussion with the following example. Consider the CAD schema. The \( MF \) for the whole CAD design is:
\[ MF(Design) = K_1 \times OP_1 ( f_{Design-Team} \times MF(Team), f_{Design-Composite} \times MF(CompositePart) ) \]

with

\[ MF(Team) = K_2 \times OP_2 ( f_{Team-Engineer} \times MF(Engineer), f_{Team-Office} \times MF(Office) ), \]

\[ MF(CompositePart) = K_3 \times OP_3 ( f_{CompositePart-Documentation} \times MF(Documentation), f_{CompositePart-AtomicPart} \times MF(AtomicPart) ), \]

\[ MF(AtomicPart) = f_{AtomicPart-Connection} \times MF(Connection), \]

with \( MF(Engineer) = 1, MF(Office) = 1, MF(Documentation) = 1 \) and \( MF(Connection) = 1 \).

The constants \( K_1, K_2 \) and \( K_3 \), and operations \( OP_1, OP_2 \) and \( OP_3 \) are determined by the nature of the composite references in the OODB schema. In particular, for the unconstrained pair-up case (both exclusive and shared):

\[ MF(Design) = f_{Design-Team} \times f_{Team-Engineer} \times f_{Team-Office} \times f_{Design-CompositePart} \times f_{CompositePart-Documentation} \times f_{CompositePart-AtomicPart} \times f_{AtomicPart-Connection}. \]

Equation 1

As all the forward fan-outs contribute towards the \( MF \), for an OODB schema with unconstrained pair-up composite references, we need to take into consideration both the "depth" and the "breadth" of the schema tree, meaning that: (a) the deeper the schema tree; and (b) the wider the schema tree, the larger the number of forward fan-outs we need to include in the \( MF \).

For the constrained pair-up and exclusive case:

\[ MF(Design) = \max( f_{Design-Team} \times \max( f_{Team-Engineer}, f_{Team-Office} ), f_{Design-CompositePart} \times \max( f_{CompositePart-Documentation}, f_{CompositePart-AtomicPart} \times f_{AtomicPart-Connection} ) ). \]

Equation 2

As only the forward fan-outs along the "maximal" path (with maximum forward fan-outs) contribute towards the \( MF \), for an OODB schema with constrained pair-up and exclusive composite references, we need only to take into consideration the "depth" of the schema tree, meaning that: the deeper the schema tree, the larger the number of forward fan-outs we need to include in the \( MF \).

For the constrained pair-up and shared case:

\[ MF(Design) = K_1 \times \max( f_{Design-Team} \times K_2 \times \max( f_{Team-Engineer}, f_{Team-Office} ), f_{Design-CompositePart} \times K_3 \times \max( f_{CompositePart-Documentation}, f_{CompositePart-AtomicPart} \times f_{AtomicPart-Connection} ) ). \]

Equation 3

**Example 2**

Let us illustrate the above cost formulae with the SJL shown in Figure 50. The SJL involves the classes \( Design \), \( CompositePart \) and \( AtomicPart \). The index tuple length is:

\[ STJI = 3 \times SIOID + SM. \]

The number of index tuples is

118
\[ n = \|Design\| \times f_{Design-CompositePart} \times f_{CompositePart-AtomicPart}, \quad \text{where} \quad f_{Design-CompositePart} \quad \text{and} \quad f_{CompositePart-AtomicPart} \] are the forward fan-out factor from class Design to class CompositePart and class CompositePart to AtomicPart, respectively. Finally, the storage cost is given by:

\[ m = \left( \|Design\| \times f_{Design-CompositePart} \times f_{CompositePart-AtomicPart} \right) \times \left( 3 \times SOID + SM \right) \times \frac{PS \times POF}{PS \times POF}. \] For a backward query, \( n \) can be similarly calculated.

\[ \Box \]

**Example 3**
Consider the SJI shown in Figure 52, the SJI involves the classes Design, Engineer, Office, Documentation and AtomicPart. To illustrate, we use Equation 3 (i.e., the constrained pair-up and shared case’s formula) to calculate the MultiplyingFactor. The index tuple length is \( STJI = 5 \times SOID + SM \). For a forward query, the number of index tuples is:

\[ n = \|Design\| \times K \times \max( f_{\text{fwd}(Design, Engineer, 1)}, f_{\text{fwd}(Design, Office, 1)}, f_{\text{fwd}(Design, Documentation, 1)}, f_{\text{fwd}(Design, AtomicPart, 1)} ), \]

where \( K \) is a small constant,

\[ f_{\text{fwd}(Design, Engineer, 1)} = \|Engineer\| \times \left( 1 - \left( 1 - \frac{f_{\text{fwd}(Design, Team, 1)}}{\|Engineer\|} \right) \right), \]

and

\[ f_{\text{fwd}(Design, Team, 1)} = \|Team\| \times \left( 1 - \left( 1 - \frac{f_{\text{fwd}(Design, Team, 1)}}{\|Team\|} \right) \right) = f_{Design-Team}. \] Similarly, \( f_{\text{fwd}(Design, Office, 1)}, f_{\text{fwd}(Design, Documentation, 1)} \) and \( f_{\text{fwd}(Design, AtomicPart, 1)} \) can also be calculated. (For details on the \( f_{\text{fwd}} \) function, refer to Appendix B). Finally, the storage cost is given by

\[ m = \left[ n \times (5 \times SOID + SM) \right] \times \frac{PS \times POF}{PS \times POF}. \]

\[ \Box \]

**5.2.4 Index retrieval cost**

For the index retrieval cost, there are two cases: Clustered index and Non clustered index.

**Clustered index**

Similar to Chapters 3 and 4, for estimating the number of disk accesses to a collection of leaf/non leaf nodes, we still use the Yao function [78]. Specifically, given \( n \) nodes uniformly distributed into \( m \) pages \((1 < m \leq n)\), if \( k \) nodes \((k \leq n)\) are randomly selected from these \( n \) nodes, the expected number of disk accesses is given by:

---

1. As in [74], we assume a two-level clustered B* tree index; for B* tree index with more levels, readers are referred to [74] for details. Higher level B* tree indices can be incorporated in our cost model easily.

119
\[ Y_{ao}(k, m, n) = m^n \left[ 1 - \prod_{i=1}^{k} \frac{n_{d-i} - i + 1}{n_{d-i} - i + 1} \right], \text{ where } d = 1 - \frac{1}{m}. \] (The expected number of disk accesses is not equal to \( k \) because some pages may contain 2 or more nodes.) But the \( Y_{ao} \) function only applies when \( m \leq n \), i.e., when a node is smaller than or equal to the page size. For larger nodes, we estimate the number of disk accesses by a simple proportion: \( m \times k/n \). In building the cost model, we use an auxiliary function \( Y \) as defined below:

\[
Y(k, m, n) = \begin{cases} 
Y_{ao}(k, m, n) & \text{Case 1: for node size smaller than or equal to page size} \\
 m^*k/n & \text{Case 2: for node size larger than page size}
\end{cases}
\]

For an SJI as shown in Figure 55 occupying \( m \) pages of storage and with \( n \) tuples, if we want to retrieve \( k \) tuples out of these \( n \) tuples, the index retrieval cost is then:

Index retrieval cost = Disk access for leaf-level nodes + Disk access for non leaf-level nodes.

The two disk access costs are calculated as follows:

- Disk access for leaf-level nodes is \( Y(k, m, n) \). (Note that the leaf-level nodes store the SJI tuples.)

- Disk access for non leaf-level nodes is \( Y(Y(k, m, n), \left\lfloor \frac{m}{B^T P} \right\rfloor, m) \).

**Non-Clustered index**

For non-clustered index, the index retrieval cost is

Index retrieval cost = Disk access for leaf-level nodes of the clustered index + Disk access for non leaf-level nodes of the non-clustered index.

Note that non-clustered index does not have its own SJI tuples, rather it shares the leaf-level nodes with the clustered index. The two disk access costs are calculated as follows:

- Disk access for leaf-level nodes of the clustered index is \( Y(k, m, n) \).

- Disk access for non leaf-level nodes of the non-clustered index is

\[ Y(k, \left\lfloor \frac{n}{B^T P} \right\rfloor, n) + Y(Y(k, \left\lfloor \frac{n}{B^T P} \right\rfloor, n), \left\lfloor \frac{n}{B^T P} \right\rfloor, \left\lfloor \frac{n}{B^T P} \right\rfloor). \]

1. As in [74], we use a three-level non-clustered \( B^+ \) tree index.

120
Example 4
Continuing from example 2, for the SJI as shown in Figure 52, the SJI has a clustered index on OIDs of class Design, and non-clustered indices on some other OIDs. There are \( n \) SJI tuples with \( n = |Design| \times MultiplyingFactor \). If the selectivity of the predicate on class Design is \( Sel \), then there will be \( |Design| \times Sel \) number of Design objects satisfying the predicate, hence the number of relevant SJI tuples retrieved is \( k = |Design| \times Sel \times MultiplyingFactor = n \times Sel \). Using the above calculated \( m \), \( n \) and \( k \), the index retrieval cost using the clustered index is calculated by:

\[ Y(k, m, n) + Y(Y(k, m, n), \left\lceil \frac{m}{BTP} \right\rceil, m) \]

If there is also a non-clustered index on the OIDs of class Documentation, the storage cost \( m \) and the total number of SJI tuples \( n \) remain the same, as the non-clustered index shares the leaf-level nodes with the clustered index. The predicate is on the root class Design, and the number of relevant SJI tuples we need to retrieve is approximated by

\[ k = n \times \frac{\text{ref}_{\text{Documentation}}}{|\text{Documentation}|} \]  
where \( \text{ref}_{\text{Documentation}} = \text{fwd}(Design, \text{Documentation}, |Design| \times Sel) \) is the number of objects referenced in class Documentation if we start traversing from the root class Design. Using this new \( k \), the index retrieval cost for this non-clustered index is:

\[ Y(k, m, n) + Y\left(k, \left\lceil \frac{n}{BTP} \right\rceil, n\right) + Y\left(Y\left(k, \left\lceil \frac{n}{BTP} \right\rceil, n\right), \left\lceil \frac{n}{BTP} \right\rceil, \frac{n}{BTP} \right) \]

5.2.5 Index maintenance cost
Using the SJI shown in Figure 55, the cost model for index maintenance for the clustered/non-clustered index is formulated. There are three basic types of index maintenance: (1) update, (2) insertion, and (3) deletion. As the cost models for the three types of index maintenance are quite similar, we concentrate below on the update cost model.

Figure 61 shows a case where there is an object update in a class \( c_i^o \). Let object \( o_i^o \) (of class \( c_i^o \)) have \( o_{i+1}^o \) (of class \( c_{i+1}^o \)) as its complex attribute value, and after the update, \( o_{i+1}^o \) be updated to \( o_{i+1}^o \) (of class \( c_{i+1}^o \)). The change in the connections of the objects thus impacts the SJI.
When there are updates to the classes, we need to update the SJI tuples (in the leaf-level nodes) as well as the index entries of the SJI (in the non leaf-level nodes). There are 3 cases:

- Case (A) -- the SJI is clustered on the class $c^j_{i+1}$.
- Case (B) -- there is a non-clustered index on the class $c^j_{i+1}$.
- Case (C) -- there is no clustered/non-clustered index on OIDs of class $c^j_{i+1}$. We need to traverse using pointer traversal towards the class upon which a clustered index is built. After that, we make use of this clustered index to facilitate the update.

If there are $u$ updates in the object connections between two neighbouring classes (say, CompositePart and AtomicPart), this will propagate to a total of $d$ updates on the SJI index tuples. We estimate $d = u \times UpdateMultiplyingFactor$ where $UpdateMultiplyingFactor$ is similar to the $MultiplyingFactor$ except that the effect of the forward fan-out (between the classes $c^j_i$ and $c^j_{i+1}$) from the formula of calculating the $MultiplyingFactor$ should be removed.

Based on the work of [74], there are four update cost components as shown below:

$$\text{Update cost} = \text{Disk accesses for pointer traversal (if required)} +$$
$$\text{Disk accesses to update the leaf-level nodes of the clustered index} +$$
$$\text{Disk accesses to update the non leaf-level nodes of the clustered index} +$$
$$\text{Disk accesses to update the non leaf-level nodes of the non-clustered index}.$$  

**Disk accesses for pointer traversal**

This is required only for Case (C) where there is no clustered/non-clustered index on the OIDs of class $c^j_{i+1}$. Let the SJI have a clustered index on the root class $c_0$. Given $u$ OIDs of class $c^j_{i+1}$, we need to traverse backward from class $c^j_{i+1}$ to the root class. We then search the clustered index on the root class to retrieve the SJI tuples related to these $u$ objects. The cost is derived using [37] as

$$\sum_{j=1}^{0} \gamma(bwd(j, j+1, u, p), ||c^j_j||, ||c^j_j||)$$

if there are reverse pointers along the path. (Refer to Appendix...
B for details on the \( \text{bwd} \) function.) If there is no reverse pointer, we need to scan all the objects in all the classes, the cost is thus \( \sum_{j=i}^{0} |c_j^p| \). Note that if there is no reverse pointer, the backward traversal cost will be much higher.

**Disk accesses to update the leaf-level nodes of the clustered index**

We need \( y_1 = Y(nst, m, n) \) disk accesses to retrieve \( nst \) number of SJI tuples from the leaf-level nodes of the clustered index and a further \( y_2 = Y(d, m, n) \) disk accesses to write back the SJI tuples actually affected by this update. Let \( C_1 \) be the class on which the clustered index is defined, then

\[
nst = u \times \frac{n}{ref_{C_1}}
\]

is the estimated number of SJI tuples that we need to retrieve, where \( ref_{C_1} \) is the number of objects of class \( C_1 \) that are relevant to the SJI.

**Disk accesses to update the non leaf-level nodes of the clustered index**

We need \( Y(y_1, \left\lceil \frac{m}{BTP} \right\rceil, m) \) disk accesses to read in the index entries of the non leaf-level nodes of the clustered index and \( Y(y_2, \left\lceil \frac{m}{BTP} \right\rceil, m) \) disk accesses to write back the actually affected clustered index entries.

**Disk accesses to update the non leaf-level nodes of the non-clustered index**

Let \( y_3 = Y(nst_1, \left\lceil \frac{n}{BTF} \right\rceil, n) \) and \( C_J \) be the class on which the non-clustered index is defined, then

\[
nst_1 = u \times \frac{n}{ref_{C_J}}
\]

is the estimated number of SJI tuples that we need to retrieve, where \( ref_{C_J} \) is the number of objects of class \( C_J \) that are relevant to the SJI. Furthermore, let \( y_4 = Y(2 \times d, \left\lceil \frac{n}{BTP} \right\rceil, n) \), where the factor 2 is due to the fact that for a clustered index we need only to amend the index entries. But in the case of a non-clustered index, we need to delete and insert back the index entries.

Thus to update the non leaf-level nodes of the non-clustered index, we need

\[
y_3 + Y(y_3, \left\lceil \frac{n}{BTF} \right\rceil, \left\lceil \frac{n}{BTF} \right\rceil) \text{ disk accesses to read the index entries of the non leaf-level nodes of the non-clustered index, and}
\]

\[
y_4 + Y(y_4, \left\lceil \frac{n}{BTF} \right\rceil, \left\lceil \frac{n}{BTF} \right\rceil) \text{ disk accesses to write back the actually affected non-clustered index entries. Note that we also need to apply these cost formulae to all non-clustered indices that are affected by these updates.}
\]
Example 5
Continuing with Examples 3 and 4, for the same SJL as shown in Figure 52, we recall that 
the SJL has a clustered index on class Design, and a non-clustered index on class Docu-
mentation. We illustrate with u updates towards the two adjacent classes CompositePart 
and AtomicPart. This is an instance of Case (C) as AtomicPart has neither a clustered, nor 
a non-clustered index defined on it. The m and n have already been calculated in Example 
3 and are re-capped below:

\[ n = \left[ \text{Design} \right] \times K \times \max \left( \text{fwd}(\text{Design, Engineer, 1}), \text{fwd}(\text{Design, Office, 1}), \text{fwd}(\text{Design, Documentation, 1}), \text{fwd}(\text{Design, AtomicPart, 1}) \right), \]

where m = \left[ \frac{n \times (5 \times SOID + SM)}{PS \times POF} \right].

Here, d = u \times K \times \max \left( \text{fwd}(\text{Design, Engineer, 1}), \text{fwd}(\text{Design, Office, 1}), \text{fwd}(\text{Design, Documentation, 1}), \text{fwd}(\text{Design, AtomicPart, 1}) \right).

Note that the last term of d is fwd(Design, CompositePart, 1), not fwd(Design, AtomicPart, 1) as found in the formula of n above, this is because we need to remove the effect of the forward fan-out from class CompositePart to AtomicPart. Furthermore, we estimate \( n_{st} \) by \( u \times \frac{n}{ref_{design}} \) and, \( n_{st1} \) by \( u \times \frac{n}{ref_{documentation}} \) for the clustered and non-clustered index, respectively. Fi-

nally, the costs for the four components are as follows:
(1) Disk access for pointer traversal:

- \( Y(bwd(\text{CompositePart, AtomicPart, u, 4}), \left\{ \text{CompositePart} \right\}, \left\{ \text{CompositePart} \right\}) + 
  \ Y(bwd(\text{Design, AtomicPart, u, 4}), \left\{ \text{Design} \right\}, \left\{ \text{Design} \right\}) \) if there are reverse pointers\(^1\);

- \( \left\{ \text{CompositePart} \right\} + \left\{ \text{Design} \right\} \) if there is no reverse pointer.

(2) Disk access to update the leaf-level nodes of the clustered index: \( Y(nst, m, n) + Y(d, m, n) \).

(3) Disk access to update the non leaf-level nodes of the clustered index:

\[ Y\left( Y(nst, m, n), \left[ m \right]_{BTF}, m \right) + Y\left( Y(d, m, n), \left[ m \right]_{BTF}, m \right). \]

(4) Disk access to update the non leaf-level nodes of the non-clustered index:

\[ Y(nst1, \left[ n \right]_{BTF}, n) + Y\left( nst1, \left[ n \right]_{BTF}, n \right), \left[ n \right]_{BTF}, \left[ n \right]_{BTF}, \left[ n \right]_{BTF} \] + 

\[ Y\left( 2 \times d, \left[ n \right]_{BTF}, n \right) + Y\left( 2 \times d, \left[ n \right]_{BTF}, n \right), \left[ n \right]_{BTF}, \left[ n \right]_{BTF}. \]

\[ \text{1. Note that the CompositePart and AtomicPart classes are on the 4th path in our CAD example schema.} \]

124
5.2.6 Number of Possible SJIHs

For an OODB schema with \( n \) classes, the number of possible SJIHs grows very quickly. To derive the number of possible SJIHs, we first calculate the number of possible SJIIs. As a SJIH is a set of SJIIs, we then calculate the different number of ways to construct a SJIH from the set of possible SJIIs.

For an OODB schema with \( n \) classes, as a SJI can cover two or more classes, the total number of different SJIIs is calculated by:

\[
\binom{n}{2} + \binom{n}{3} + \ldots + \binom{n}{n-1} = 2^n - n - 1
\]

where \( \binom{n}{k} \) is the number of different combinations of choosing \( k \) objects out of \( n \) objects.

As a SJIH can have either one, two, ..., up to \( n - 1 \) SJIIs in it, the total number of possible SJIHs is calculated by:

\[
\binom{2^n - n - 1}{1} + \binom{2^n - n - 1}{2} + \ldots + \binom{2^n - n - 1}{n-1}.
\]

For large \( n \), the total number of possible SJIHs approaches \( O\left(2^n\right) \). For illustration, the following table shows the number of possible SJIHs for \( n \) ranges from 3 to 8. And we note that the number of possible SJIHs grows exponentially.

<table>
<thead>
<tr>
<th>( n )</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of possible SJIHs</td>
<td>10</td>
<td>231</td>
<td>17,901</td>
<td>4,613,029</td>
<td>3,851,826,154</td>
<td>10,518,187,943,679</td>
</tr>
</tbody>
</table>

5.2.7 Remarks

Although for expository purpose we have used the CAD example schema in our above discussion, our cost model is applicable to any general OODB schema. In order to have a better understanding of the cost model and the utility of the SJIH framework, we provide in the next section some analytical experimental results to evaluate different OODB indexing methods. Our experiments are based on the cost model formulae developed above, and the results indicate that there is a wide range of queries which can benefit from SJIH.

---

1. The maximum number of SJIIs in a SJIH without redundancy is \( n - 1 \). An example of such SJIH (with \( n - 1 \) SJIIs) is the BSJH--with all \( n - 1 \) BJIIs between every two neighbouring classes. By redundancy in a SJIH, we mean that no SJI (in the SJIH) is completely contained in another SJI, which waste storage spaces and has no improvement in performance.
5.3 Evaluation of SJIH Indexing Methods

In this section, the use of SJIH is compared against pointer traversal for query processing so as to illustrate that SJIH has superior performance. We then compare the use of SJIH with other OODB indexing methods, namely, Multi-index, Nested index and Access Support Relation. The comparison is based on the storage costs and the retrieval costs for complex objects using different indexing methods. In the third experiment, we compare the utility of different SJIHs based on the index retrieval cost and update cost. Finally, in the fourth experiment, we compare the utility of different SJIHs in a large OODB schema.

5.3.1 Comparison of SJIH against Pointer Traversal

This experiment compares the utility of using different types of SJIH against pointer traversal. We further examine the effect of the variation in the object size and selectivity on the query processing costs.

Experiment setup

We use the CAD example schema (as shown in Figure 3) for this experiment to evaluate the utility of the following three SJIHs:

- Complete-SJIH (CSJIH) -- the complete SJIH, a single SJI that covers all eight classes of the schema.
- Base-SJIH (BSJIH) -- the SJIH with all the BJIIs between every pair of adjacent classes. There will be a total of seven BJIIs in the BSJIH.
- Partial-SJIH (PSJIH) -- a partial SJIH (as shown in Figure 52) that covers five classes in the schema.

The following table shows the cardinalities and forward fan-outs used in this experiment. In order to concentrate on the utility of the different SJIHs, we fix the cardinalities for all the classes. Furthermore, the forward fan-outs can be changed by varying the value of the scale factor (like in [76]). We vary the scale factor in this experiment from 0.1 to 2.0.

<table>
<thead>
<tr>
<th>Class</th>
<th>Design</th>
<th>Team</th>
<th>Engineer</th>
<th>Office</th>
<th>CompositePart</th>
<th>Documentation</th>
<th>AtomicPart</th>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cardinality</td>
<td>1,000,000</td>
<td>1,000,000</td>
<td>1,000,000</td>
<td>1,000,000</td>
<td>1,000,000</td>
<td>1,000,000</td>
<td>1,000,000</td>
<td>1,000,000</td>
</tr>
<tr>
<td>$f^p_i$</td>
<td>sf</td>
<td>sf</td>
<td>sf</td>
<td>sf</td>
<td>sf</td>
<td>sf</td>
<td>sf</td>
<td>sf</td>
</tr>
<tr>
<td>Scale factor</td>
<td>sf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.1, 0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75 and 2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From the cost model, the degrees of sharing\(^1\) between composite objects and their component objects (i.e., the forward fan-out $f^p_i$ and the reverse fan-out $r^p_{i+1}$ values) are the most influ-
ential factors on the index retrieval cost. In this experiment, we set both $f_i^P$ and $r_{i+1}^P$ to $sf$ -- the scale factor. We can observe the effect of the variations in the degree of sharing on the performance of different SJIHs by varying the scale factor. For uniform comparison, the cardinalities of all the eight classes are set to one million. The selectivity of the predicate of the OQL query on the root class *Design* varies from 0.01 to 1.0. The object size (in unit of bytes) varies from 500 to 128,000.

**Results and Observations**

We first present the results of the comparison with respect to the variation in object size. We examine the complex object retrieval that requires the five classes in Figure 52. As the results for PSJIH are similar to CSJIH /BSJIH, in Figures 62 and 63, we only show the plots of Object size vs. Cost ratio for CSJIH and BSJIH. We concentrate on scale factor range: 0.5, 1.0 and 2.0. The performance metric is the Cost Ratio (CR) which is defined as:

$$ CR = \frac{\text{Cost of using SJIH for complex object retrieval}}{\text{Cost of using Pointer Traversal for complex object retrieval}}. $$

Note that a CR value of less than 1 implies that SJIH is more beneficial. The observations from this experiment are summarized as follows:

1. Regarding the cost ratios, most of the CR values in Figures 62 and 63 are much smaller than 0.1, implying that the use of SJIH is very much beneficial when compared to pointer traversal. The use of SJIH drastically reduces the query processing cost.

2. To observe the effect of variations of object size on the cost ratio, we note that there are two regions in Figure 62:

   - For object size smaller than or equal to the page size of the system (8KB): when the object size is close to the page size, the CR decreases slowly with the increase of object size. The trend can be exemplified by the results with scale factor of 1.0 for CSJIH. The CR values for the different object sizes (SC) are:

     | SC   | 500  | 1,000 | 2,000 | 4,000 | 8,000 | 16,000 | 32,000 | 64,000 | 128,000 |
     |------|------|-------|-------|-------|-------|--------|--------|--------|---------|
     | CR   | 0.02272 | 0.01171 | 0.00853 | 0.00586 | 0.00408 | 0.00204 | 0.00102 | 0.00051 | 0.00026 |

   - For object size larger than the page size: the CR decreases more rapidly with the increase of object size. From the cost model, if the object size is larger than the page size, CR will be inversely proportional to the object size. One conclusion we can draw is that large object size favours the use of SJIH.

From the cost model, the index retrieval cost (cf. section 5.2.4) is independent of the object

---

1. In the following discussion, we will use the term "degree of sharing" to mean both the forward fan-out and the reverse fan-out values.

127
size. The decrease in CR is due to the increase in the pointer traversal cost. For region 1, where the object size is smaller than the page size of the system, the pointer traversal cost increases when the object size increases. But due to the effect of buffering of objects which is a property of the Yao function (cf. section 5.2.4), the pointer traversal cost will not increase as fast as the object size, which leads to the slow decrease in the CR when object size is close to the page size of the system. For region 2, the object size is larger than the page size. In this case, one page of main memory cannot hold more than one object, and the Case 2 of the Yao function (cf. section 5.2.4) predicts that the pointer traversal cost will be proportional to the object size. Therefore the CR is inversely proportional to the object size.

(3) When comparing the two sets of results for CSJIH and BSJIH, we observe that the CRs of CSJIH are more sensitive to the variation of the scale factor. The performance of CSJIH at scale factor 0.5 is much better than that of scale factor 2.0. From the cost model, the index retrieval cost of a complex SIJIH is highly dependent on the total number of index tuples in the SIJIH. This is equal to the number of object connections between the classes. And the number of object connections is dependent on the forward fan-out values among all the involved paths (cf. section 5.2.3). High scale factor implies high forward fan-out values, and hence high index retrieval cost. This explains why the CR at scale factor 2.0 is poorer than 0.5. The conclusion here is that low degree of sharing favours the use of more complex SIJIH.

For the BJIs in the BSJIH, the multiplying effect of the forward fan-out value does not have a prominent effect on the index retrieval cost. Hence the index retrieval costs are not so sensitive to the change in the degree of sharing. The conclusion is therefore to use simpler SIJIH for high degree of sharing.

We now present the results of comparison with respect to the variation in selectivity. In Figures 64 and 65 we show the plots of Selectivity vs. Cost ratio for CSJIH and BSJIH. The other parameter settings are the scale factor which varies between 0.5, 1.0 and 2.0, and the object size which is 4,000 bytes. The observations for these results are as follows:

(1) On the effect of variations of selectivity over the cost ratio: CR decreases as selectivity increases. As explained in Appendix C, when the selectivity is higher than a particular threshold, the index retrieval cost will be independent of the selectivity. The reason for the decrease in CR when selectivity increases is due to the pointer traversal query processing cost. For pointer traversal, higher selectivity means more objects will be retrieved, implying higher query processing cost and hence the CR decreases. This leads to the conclusion that high selectivity favours the use of SIJIH instead of pointer traversal.

(2) When comparing the two result plots in Figures 64 and 65, we observe similar trends as in the previous two figures (Figures 62 and 63): the CRs of more complex SIJIH are more sensitive to the
variation in the scale factor. Depending on the degree of sharing between classes in an OODB schema, CSJIH can result in either better or poorer results, therefore a judicious selection of the set of SJIIs in an SJIH is important.

Figure 62: Object size vs. Cost ratio for CSJIH

Figure 63: Object size vs. Cost ratio for BSJIH
Figure 64: Selectivity vs. Cost ratio for CSJH

Figure 65: Selectivity vs. Cost ratio for BSJH
5.3.2 Comparison of Different Indexing Methods

In this second experiment, we compare Multi-index, Nested index, Access Support Relation and our SJIH. The comparison is based on the storage cost and the index retrieval cost for complex object retrieval.

Experiment setup

We use the CAD schema as in the previous experiment and concentrate on the complex object retrieval as shown in Figure 52. Note that:

- For Multi-index, we build a total of six MIs to associate every two neighbouring classes.
- For Nested index, we build four NIs, one NI for each of the following class pairs: Design-Engineer, Design-Office, Design-Documnetation, and Design-AtomicPart.
- For ASR, we build four ASRs, one ASR for each of the following class groups: Design-Team-Engineer, Design-CompositePart-AtomicPart, Team-Office, and CompositePart-Documnetation.
- For SJIH, we use the three SJIHs as in section 5.3.1.

In this experiment, the scale factor range is: 1/16, 1/8, 1/4, 1/2, 1, 2, 4, 8 and 16. The other parameter settings are the same as in the previous experiment.

Results and Observations

In Figure 66, we show the plot of Scale factor vs. Storage cost for the different indexing methods. We observe the following:

(1) At low scale factor, PSJIH and CSJIH are the best in terms of storage cost requirement. As the scale factor increases, the storage costs of all the indexing methods also increase. As the scale factor further increases, MI and BSJIH become the best while CSJIH's storage cost keeps increasing and becomes the highest storage cost.

Note that a low scale factor means the forward fan-out values are also low, which implies that there are very few object connections between the objects in classes Design, Engineer, Office, Documentation and AtomicPart. Due to the multiplying effect in the forward fan-out values along the paths, the more complex SJIHs have the lower storage cost at low scale factor. As MI and NI involve only one path in the schema, the multiplying effect in the forward fan-out values is not as prominent as that on the multi path SJIHs, so they will have higher storage cost at low scale factor. But when the scale factor is larger than 1, the multiplying effect in the number of object connections causes more complex SJIHs' storage costs to increase rapidly. Our conclusion is that since BJI and DJI/HJI serve similar functionality as Multi-index and Nested index, we should employ SJIH with simpler BJIs, DJIs and HJIs in cases of high degree of sharing.

(2) In Figures 67 to 70, we show the plots of Scale factor vs. Index retrieval cost for the different
indexing methods with selectivities at 0.001, 0.01, 0.1 and 1.0, respectively. We focus on Figure 67 first. From the cost model, the index retrieval cost (cf. section 5.2.4) has a high correlation with the storage cost. Hence we see a very similar pattern for the variations of the index retrieval cost and the storage cost as in Figure 66.

(3) Next we compare the four result plots in Figures 67 to 70. As explained in Appendix C, when the selectivity is higher than a particular threshold, the index retrieval cost will be independent of the selectivity. As selectivities of 0.1 and 1.0 are above the threshold, the result plots in Figures 69 and 70 are identical, showing that the index retrieval cost is independent of the selectivity.

From Figures 67 to 70, one can notice that MI, NI and BSJIH indexing methods are unsuitable for scale factors less than 1, because there are other indices (PSJIH and CSJIH) which outperform MI/NI by a factor of 100. For large scale factors, the only SJIIH that provides as good performance as MI/NI is BSJIH. Depending upon the scale factor, we can decide on the SJIIH one should consider.

![Plot of Scale factor vs. Storage cost](image)

Figure 66: Scale factor vs. Storage cost
Figure 67: Scale factor vs. Index retrieval cost at selectivity 0.001

Figure 68: Scale factor vs. Index retrieval cost at selectivity 0.01
Figure 69: Scale factor vs. Index retrieval cost at selectivity 0.1

Figure 70: Scale factor vs. Index retrieval cost at selectivity 1.0
5.3.3 Comparison of Index retrieval cost against Update cost

Experiment Setup

We now compare the index retrieval cost with the update cost using the CAD example schema shown in Figure 3. We still use the three SJIHs as in the previous two experiments. The experiment illustrates the update costs for 100, 10,000, and 1,000,000 updates between the classes CompositePart and AtomicPart, respectively, as discussed in Example 5 and with reverse pointers available for the backward pointer traversal (cf. section 5.2.5). Furthermore, we illustrate the index retrieval costs for the complex object retrieval as shown in Figure 52 (with selectivity of 0.01). Our main concerns are on the combined effect of the complex object retrieval and the updates. The total cost is given by:

\[
Total\ cost = p \times Update\ cost + (1 - p) \times Index\ retrieval\ cost
\]

where \( p \) is the update proportion. An update proportion of 1.0 means no retrieval, while an update proportion of 0.0 means no update. In these experiments we vary \( p \) from 0.0 to 1.0, and the number of updates vary from 100 to 1,000,000.

Results and Observations

With 100 updates, for the three different SJIHs, we plot Update proportion vs. Total cost in Figures 71 to 74. The scale factor varies between 0.5, 1.0, 2.0 and 4.0 for Figures 71 to 74, respectively. The observations are: (1) as the scale factor increases, the total cost for all three SJIHs also increases; (2) at scale factor 0.5, CSJIH is the best, PSJIH the second, and BSJIH is worst; (3) at scale factor 1.0, PSJIH is just a bit better than CSJIH, and BSJIH is the worst; (4) at scale factor 2.0 and update proportion less than 0.9, PSJIH is the best, BSJIH is the second, and CSJIH is the worst;\(^1\) and (5) at scale factor 4.0, BSJIH is the best, PSJIH is the second, and CSJIH is the worst.

This can be explained by the index retrieval and update cost model (cf. sections 5.2.4 and 5.2.5):  

- In Figure 71 with low degree of sharing/scale factor 0.5, we can predict from the cost models that:

  IRC(BSJIH) > IRC(PSJIH) > IRC(CSJIH), where IRC() is the index retrieval cost and
  UC(BSJIH) > UC(PSJIH) > UC(CSJIH), where UC() is the index update cost.

  And in Figure 71, as the index update costs (for 100 updates) for the three SJIHs are relatively smaller than the index retrieval costs, this explains the decreasing trend of all the three curves.

- In Figure 74 with high degree of sharing/scale factor 4.0, we can also predict from the cost models that:

\(^1\) If the update proportion increases beyond 0.9, BSJIH will be the best, PSJIH the second, and CSJIH the worst.
IRC(CSJH) > IRC(PSJIH) > IRC(BSJIH) and
UC(CSJH) > UC(PSJIH) > UC(BSJIH).

And in Figure 74, as the index update costs (for 100 updates) for the three SJIHs are relatively smaller than the index retrieval costs, this also explains the decreasing trend of all the three curves.

Regarding the use of SJIH in a mixed environment of update and complex object retrieval, if the number of updates is small (100) when compared with the cardinalities of the classes (1,000,000) in the OODB schema, the recommendation from the above experiment is as follows:

- At low scale factor (0.5), CSJIH is useful.
- At medium scale factor (1.0), both PSJIH and CSJIH are useful.
- At high scale factor (2.0 and 4.0), BSJIH is useful.
Figure 72: Update proportion vs. Total cost for Scale factor of 1.0

Figure 73: Update proportion vs. Total cost for Scale factor of 2.0
With 10,000 updates and for the three different SJIHS, we plot Update proportion vs. Total cost, and scale factor varies between 0.5 and 4.0 in Figures 75 and 76, respectively. Similarly, with 1,000,000 updates, we also plot Update proportion vs. Total cost, and scale factor varies between 0.5 and 4.0 in Figures 77 and 78, respectively. The observations from these plots are:

- As the number of updates increased from 100 (as in the previous set of result plots) to 10,000 and 1,000,000, the index update cost increases. The index update costs for CSJIH and PSJIH are comparable/higher than that of the index retrieval costs. In Figures 75 to 78, we observe an increasing trend in the curves for CSJIH and PSJIH. The conclusion is that in case where there is large number of index updates in the application environment, CSJIH and PSJIH will not be too useful.

- Even at high degree of sharing/scale factor, the index update costs for BSJIH are relatively small when compared with that of CSJIH and PSJIH. Further, in Figures 75 to 78, we observe a decreasing trend in the curves for BSJIH. The conclusion is that in case where there is large number of index updates in the application environment, BSJIH can still be used to expedite complex object retrievals.
Figure 75: For Scale factor of 0.5 and 10,000 Updates

Figure 76: For Scale factor of 4.0 and 10,000 Updates
5.3.4 Comparison of Different SJIHs in a Large OODB Schema

In this fourth experiment, we consider retrieval of complex objects from an OODB schema with forty classes, which is more complex than the previous experiments. The comparison is based on the storage cost and the index retrieval cost for complex object retrieval.
Experiment setup

We consider a database schema with a class composition hierarchy with a total of 40 classes as shown in Figure 79. Note that this schema is more realistic than the OODB schema used in the previous experiments. The aim of using such a schema is to observe the trend of how SJIH behaves in cases where the number of classes accessed varies from small (2 classes) to large (40 classes).

Figure 79: Large Database Schema with 40 classes

For the indexing methods, we have the following six SJIHS:

- **BSJIH** - the set of all BJIIs between every adjacent pair of classes in the OODB schema;
- **CSJIH** - the SJIH with a single SJI that covers all 40 classes in the OODB schema;
- **PSJIH1, PSJIH2, PSJIH3, PSJIH4** - as shown in Figures 80 to 83, these four PSJIHs are arranged in order of complexity\(^1\): with PSJIH1 being quite similar with the BSJIH (PSJIH1 has 27 BJIs); PSJIH2 is more complex than PSJIH1; PSJIH3 is even more complex than PSJIH2; and finally PSJIH4 consists of 4 complex SJIs.

To concentrate on the impacts of the variations of the forward/backward fanouts on the storage cost and index retrieval cost, we set all the forward/backward fanouts in all the links in the OODB schema to \(sf\) -- the scale factor. In this experiment, we randomly select the classes to be accessed. We generate queries randomly (total 100 queries), execute them and calculate the number of disk accesses required by the complex object retrievals. We present the mean number of disk accesses (over the 100 queries). We repeat the above for each of the six SJIHS. Further, we have the following six types of query: Q2, Q5, Q10, Q20, Q30 and Q40 that access 2, 5, 10, 20, 30 and 40 classes respectively. The classes are selected randomly from the class composition hierarchy and the leaf nodes are selected from the subtree of the previously selected upper level node.

\(^1\) The degree of complexity is measured by the number of classes covered by the SJIIs in the SJIH. The larger the number of classes covered by the SJIIs, the more complex is the SJIH.
Results and Observations

Storage cost

In Figure 84, we show the plot of Scale factor vs. Storage cost for the different indexing methods. We observe the following:

(1) At low to medium scale factor (0.1 - 1.0), CSJIH is the lowest in terms of storage cost required, implying that CSJIH is the best candidate as the indexing method in this scale factor range. The storage costs are in the following order:

\[
\text{BSJIH} > \text{PSJIH1} > \text{PSJIH2} > \text{PSJIH3} > \text{PSJIH4} > \text{CSJIH} \quad \text{(A)}
\]

The explanation for the ordering in (A) is as follows:

- as we are using the constrained pair-up and exclusive case’s formula, the storage cost \( m \) is proportional to number of index entry and the size of each index entries, i.e., \( m \propto n \times size \), where \( n \) is the number of index entry, \( size \) is the size of each index entries.
- for BSJIH, \( m \propto 39 \times sf \times 2 \) (as there are 39 BJIs, and the number of index entries is proportional to \( sf \) - the degree of sharing, and there are 2 OID\(s \) in each index entry).
- for PSJIH1, \( m \propto (4 \times sf \times 4 + 27 \times sf \times 2) \) (as there are 4 SJI\(s \) (each with 4 OId\(s \)) and 27 BJIs, refer to Figure 80 for details).
- for PSJIH2, \( m \propto (13 \times sf \times 4) \).
- for PSJIH3, \( m \propto (sf^2 \times 13 + 9 \times sf \times 4) \).
- for PSJIH4, \( m \propto (sf \times 4 + 3 \times sf^2 \times 13) \).
- for CSJIH, \( m \propto sf^3 \times 40 \).
- when \( sf \) is less than or equal to 1.0, for CSJIH -- \( m \propto sf^3 \times 40 \), this implies that the storage cost for CSJIH is the lowest.
- and hence it explains (A) -- the order of storage cost is \( \text{BSJIH} > \text{PSJIH1} > \text{PSJIH2} > \text{PSJIH3} > \text{PSJIH4} > \text{CSJIH} \).
(2) As the scale factor increases, the storage costs of all the indexing methods also increase. At high scale factor (say 2.0), CSJIH’s storage cost becomes the highest. This implies that at high scale factor ranges (say >1.0), CSJIH will not be the best indexing method and some other simpler SJIH is more suitable (say PSJIH2). At scale factor 2.0, the storage cost is in this order:

\[
\text{CSJIH} > \text{PSJIH4} > \text{BSJIH} > \text{PSJIH1} > \text{PSJIH3} > \text{PSJIH2}
\]  

(B)

The explanation for the ordering in (B) is similar to (A), but

- this time the degree of sharing/scale factor is larger than 1.0, which implies that the storage cost for CSJIH is the highest.
- the above storage cost formula explains (B) -- the order of storage cost is CSJIH > PSJIH4 > BSJIH > PSJIH1 > PSJIH3 > PSJIH2.
Index retrieval cost

In Figures 85 to 90, we show the result plots for Scale factor vs. Cost ratio for the different types of queries, namely: Q2, Q5, Q10, Q20, Q30 and Q40.

The observations are:

(1) In Figure 85, for Q2 (accessing 2 classes), we observe that except for very low scale factor (0.1 - 0.25), BSJIH is the best in terms of cost of complex object retrieval. The explanation is:

- The index retrieval cost (IRC) is highly dependent on the index storage cost (SC). As discussed in the previous section on storage cost, we observe that at very low scale factor (0.1-0.25), CSJIH has the lowest SC and hence it also has the lowest IRC. But Figure 85 is for queries accessing only 2 classes, therefore as the degree of sharing increases, the use of CSJIH (which contains all 40 OIDs of all classes in the OODB schema) is a waste. We only need a few simple BJs to facilitate the retrieval.

(2) In Figure 90, for Q40 (accessing all 40 classes), a very complex object retrieval, we observe that:

(a) for low to medium scale factor (0.1 - 1.0), CSJIH is the best; (b) for high scale factor (say 2.0), PSJIH2 is the best. The explanation is as follows:

- for (a) -- In Figure 90, we are accessing all the classes, so the use of CSJIH is not a waste, and as CSJIH has the lowest SC and hence the lowest IRC in this low to medium scale factor, hence CSJIH has the lowest IRC and is the best indexing method in a Q40 case and this range of scale factor.

- for (b) -- Although CSJIH has all the 40 OIDs required by the queries, at high scale factor (say 2.0), the SC and IRC of CSJIH will go up. This situation favors the use of a simpler SJIH with low SC. As PSJIH2 has the lowest SC at scale factor 2.0, this explains why PSJIH2 is the best indexing method in a Q40 case and this range of scale factor.
Figure 85: Scale factor vs. Cost ratio for Q2 (accessing 2 randomly selected classes)

Figure 86: Scale factor vs. Cost ratio for Q5 (accessing 5 randomly selected classes within the same subtree)

Figure 87: Scale factor vs. Cost ratio for Q10 (accessing 10 randomly selected classes within the same subtree)

Figure 88: Scale factor vs. Cost ratio for Q20 (accessing 20 randomly selected classes within the same subtree)
In Figures 91 to 96, we show the result plots for Scale factor vs. Cost ratio for the different types of SJIHs, namely: BSJIH, PSJIH1, PSJIH2, PSJIH3, PSJIH4 and CSJIH. The observations are:

(1) In Figure 91, for BSJIH, we observe that BSJIH is best for Q2 and worst for Q40. The explanation is:

- It is intuitive that BSJIH with BJIs (indexing on two adjacent classes) will facilitate the query processing for Q2 queries that accesses two classes. But for Q40 queries, BSJIH will not perform as well as in Q2, as Q40 queries require 40 OIDs.

(2) In Figure 96, for CSJIH, we observe that CSJIH is best for Q40 and worst for Q2. The explanation is:

- It is intuitive that CSJIH with a SJI that covers all 40 classes in the OODB schema will be the best for Q40. Further, such a complex CSJIH will be a bit wasteful for the very simple query type Q2.

(3) In Figures 91 to 95, we often observe a "dip" in the cost ratio as scale factor increases, say the curve Q2 in Figure 91 for BSJIH. This "dip" which decreases and then increases in cost ratio when scale factor increases can be explained by the following:
the curve shows the cost ratio -- the ratio between the cost of using SJIH and cost of using pointer traversal.

we observe from the cost model that: (a) the cost of using SJIH, i.e., the IRC will be highly dependent on SC and hence highly dependent on the degree of sharing/scale factor $sf$. (Note that the SCs for BSJIH, PSJIH1, and PSJIH2 are linear on $sf$; the SCs for PSJIH3 and PSJIH4 will be mainly proportional to $sf^2$; and finally, the SC of CSJIH is proportional to $sf^3$; (b) the cost of using pointer traversal (PTC) is of the form of a Yao function which has the property that if the number of object references is low (in the case of low scale factor) the value of the Yao function is also low; as the number of object references increases (i.e., the scale factor increases) the value of Yao function increases rapidly; as the number of object references further increases (and is close to the maximum -- the total number of objects in the class) the value of the Yao function becomes "saturated" and tends to a constant value (approximately the storage cost of all objects in the class).

as elaborated above, at low $sf$, the PTC is relative smaller (when compared with PTC at higher $sf$). As PTC is the denominator in the cost ratio, this explains why initially as $sf$ increases, the cost ratio decreases. At high $sf$, the Yao function is saturated and tends to a constant value, i.e., PTC stops increasing as $sf$ increases, but IRC keeps on increasing. this explain why after the dip, the cost ratio curve increases as $sf$ further increases.

(4) As another observation from Figures 91 to 96: all the cost ratio curves for BSJIH (in Figure 91) are more "flat", i.e., not so steep or the variations of cost ratio at different scale factor are not so "high"; but the cost ratio curves for CSJIH (in Figure 96) are very "steep" and the variations of cost ratio at different scale factor are high. The explanation is as follows:

- from the cost model, the SC and IRC for BSJIH are linear with $sf$, so the curves for cost ratio are not so sensitive to the variations in $sf$ (when compared with CSJIH). But the SC and IRC for CSJIH are proportional to $sf^3$, these make the cost ratio curve highly sensitive to the variations in $sf$. This explains the "flat" and "steep" nature of the curves for different SJIHs.

- The conclusion is as follow. Firstly, use CSJIH to get high cost saving if you are sure that the application environment is with either (a) low degree of sharing and/or (b) highly complex object retrieval that requires most of the classes in the OODB schema. Secondly, use simpler SJIH (say PSJIH2) if you want to have either: (a) a cost saving that is not too sensitive to the variations of degree of sharing and/or (b) a not too high variation in the cost saving in processing queries that access different numbers of classes (from Q2 all the way through Q40).
5.3.5 Summary of results

From the above analytical experiments, we have the following summary of results:

- SJIH is superior in performance to pointer traversal for complex object retrieval;
- For an OODB schema with multiple paths, if the degree of sharing along a particular portion of the schema is low, we should group the classes into a more complex SJIH for efficient complex object retrieval; and
- If the degree of sharing along a particular portion of the schema is high, we should not build a complex SJIH, but rather build a simpler SJIH with BJIs, DJIs and HJIs along that portion of the schema.

5.4 Discussion and Guidelines

In order to facilitate index designers to choose the best SJIH in different database characteristics and/or application environment, in this section we provide a summary of recommendations for the best SJIH in different situations. The recommendations are derived from the experimental results in the previous performance evaluation section.

5.4.1 Comparison of different SJIHs for complex object retrieval

In terms of storage cost (c.f. Figure 84):
• at low to medium degree of sharing (0.1 to 1.0), CSJIH is the best.
• at high degree of sharing (say 2.0), Partial-SJIH with a set of simple BJIs / DJIs / HJIs / SJIs is the best.

In terms of index retrieval cost for complex object retrieval, the IRC is highly dependent on the number of classes relevant to the complex object retrieval:

• as illustrated in Figure 85, if the number of relevant classes is small (say, Q2), CSJIH is the best at low degree of sharing (0.1 to 0.25); at medium to high degree of sharing (0.5 to 2.0), BSJIH is the best
• as illustrated in Figure 90, if the number of relevant classes is high (say, Q40): CSJIH is the best at low to medium degree of sharing (0.1 to 1.0); at high degree of sharing (2.0), PSJIH2 is the best

From the above discussion, the recommendation for best SJIH is summarized in Table 16:

<table>
<thead>
<tr>
<th>Degree of Sharing</th>
<th>Number of relevant classes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>small (2)</td>
</tr>
<tr>
<td>low (0.1)</td>
<td>CSJIH</td>
</tr>
<tr>
<td>medium (1.0)</td>
<td>BSJIH</td>
</tr>
<tr>
<td>high (2.0)</td>
<td>BSJIH</td>
</tr>
</tbody>
</table>

### 5.4.2 Guidelines for index designers

For a production type OODB system with rather static database characteristics:
• if the objective is to obtain the highest cost gain (when compared with pointer traversal), then refer to Table 16 to set up the best SJIH.

For an evolving type of OODB system with dynamic database characteristics, the degree of sharing will change as the OODB system evolves. Furthermore, the number of relevant classes in complex object retrieval required by the application environment may vary. Thus the recommendations are as follows:
• if the objective is to obtain a rather even performance gain. PSJIH2 gives the most even performance gain for all types of queries (Q2 through Q40), as illustrated in Figure 93.
• if the objective is to obtain the highest cost gain, then refer to Table 16 to set up the best SJIH.

### 5.5 Summary

Queries in OODB systems typically retrieve complex objects or their component objects. In order to support their efficient execution, it is important to provide suitable access methods for complex
object retrieval. Earlier work on OODB indexing has addressed efficient navigation through path expressions in OODBs. In this chapter, we have extended existing work by advocating a structural join index hierarchy for facilitating complex object retrieval. A structural join index hierarchy (SJIIH) is a sequence of OIDs which provides direct access to component objects of a complex object. Three types of SJIIHs have been studied in this chapter: the Complete-SJIIH provides direct access to all the OIDs of the component objects of a complex object, the Base-SJIIH requires a join of all the base join indices to retrieve the complete complex object, and the Partial-SJIIH provides direct access to the most frequently accessed component objects. Such a SJIIH framework unifies various previous indexing methods proposed for OODBs. A cost model for processing the queries and maintaining the indices is developed. Our performance results demonstrated the utility of the three SJIIHs, and showed the superiority of selecting a SJIIH over other indexing schemes, such as nested index, multi-index and access support relation.

SJIIH framework facilitate efficient processing of complex object retrievals. Example applications that can benefit from our SJIIH framework in OODB include: multimedia systems (like, Web page systems) and manufacturing engineering design (CAD/CAM) systems.

One limitation of our SJIIH framework is that if the objects in the OODB are small (and are not too large when compared with the size of an OID), then the performance gain of applying SJIIH will diminish. Further, if the selectivity is very low, that means only a very small portion of the objects in the OODB are retrieved (say, only 100 objects out of 1,000,000 objects), then the use of SJIIH will not be beneficial. Another limitation of our SJIIH framework is the storage overhead, if there is not enough storage space available for storing the SJIIH, then it will be an obstacle to apply our SJIIH technique.
Chapter 6

Optimal SJIH Algorithm

In this chapter and [32,33], we present a heuristic algorithm developed for selecting appropriate indices to efficiently process a given set of queries. Our results show that SJIH indexing mechanisms not only facilitate efficient retrieval of complex objects, but also outperform many of the existing indexing mechanisms for complex object retrieval. Moreover, given a set of queries and a limited index storage space, the heuristic algorithm facilitates fast selection of a near-optimal set of indices for efficiently executing the queries.

The rest of the chapter is organized as follows: Section 6.1 stipulates the development of a heuristic algorithm for selecting the optimal or near-optimal indices, namely Hill-Climbing Heuristic SJIH Algorithm (HCHSA). Section 6.2 presents results from HCHSA with random queries and Section 6.3 presents performance gain between the initial and final SJIH from HCHSA with random queries. Section 6.4 discusses the integration of VCP techniques into the SJIH framework and finally, Section 6.5 presents the summary of this chapter.

6.1 Hill-Climbing Heuristic SJIH Algorithm (HCHSA)

To facilitate efficient design of the optimal or near-optimal SJIH based on the input of database characteristics and query characteristics, we have developed a hill-climbing heuristic SJIH algorithm (HCHSA). The HCHSA uses the concept of unconstrained pair-up (recall from Chapter 5 that if the two branches B1 and B2 of an OODB schema are unconstrained pair-up, the objects in the two branches can be paired up freely without any constraint).

The design of the HCHSA is based on the following observations:

- As the set of queries in the query processing environment usually does not involve all the
classes in the OODB schema, the query graph (QG) has fewer or the same number of nodes as
the schema graph (SG). Dealing with a QG instead of the SG can thus potentially reduce the
search space of the HCHSA.

• Forming a SJI that covers unconstrained pair-up branches in the OG is expensive both in terms
of storage space and retrieval/update cost. We avoid this by separating unconstrained branches
from the QG. We break the QG into a number of simpler QGs: one QG contains the
constrained pair-up branches, and the other QGs contain one unconstrained pair-up branch in
each QG. By doing this, the search space is significantly trimmed down by breaking the QG
into a set of simpler QGs (SQG).

6.1.1 HCHSA
The algorithm is as follows:

Step 1: Construct the Query Graph (QG) from the Schema Graph (SG) and
the query characteristics.
Step 2: Apply branch separation to the QG and obtain a Set of simpler QGs (SQG).
The separation is guided by the information about the constrained/
unconstrained pair-up between the different branches in the QG
derived in Step 1.
Step 3: For each QG in the SQG, perform a hill-climbing heuristic process to
find the minimum cost SJIH. The process is guided by the cost model.
Step 4: Collect the SJIHs in all the SJIHs obtained in Step 3 into a new SJIH as
the final optimal or near-optimal SJIH.

Figure 97: Hill-Climbing Heuristic SJIH Algorithm

6.1.2 Illustrative example
In Figure 98 we illustrate the use of HCHSA with an extended CAD schema graph (SG). The left
side of the schema is identical to our previous CAD schema. The schema consists of a class com-
position hierarchy with six paths. In this schema, a CAD design further has a number of factories
(Factory) and suppliers (Supplier). In turn a supplier can supply a number of parts and has a num-
ber of quotations (Quotation) for the parts it supplies.

Figure 98: Example schema for HCHSA

154
The query processing environment consists of two queries as shown in Figures 99 and 100. In Step 1, we construct the QGs for each query in the query processing environment. Figure 101 shows the final QG obtained from the union of the nodes of the two QGs.

From the semantics of the CAD design in our schema, the four branches: Design-Team-Engineer, Design-Team-Office, Design-CompositePart-Documentation, and Design-CompositePart-AtomicPart-Connection, are constrained pair-up branches. Whereas, the two branches: Design-Factory and Design-Supplier-Quotation are unconstrained pair-up branches. After the branch separation in Step 2, we obtain three simpler QGs as shown in Figure 102, and we denote the resulting set of these simpler QGs by SQG.

Figure 99: Query 1

Figure 100: Query 2

Figure 101: Query Graph

Figure 102: SQG after branch separation

Figure 103: Initial SJH for the HCHSA

Figure 104: Optimal SJH for both EEA and HCHSA
In Step 3 of our HCHSA, we start with the leftmost QG, then the middle QG, and finally the rightmost QG. As the hill-climbing heuristic process requires an initial guess, we use the SJIH that just covers all the classes in the most important query\(^1\). Such an initial guess will be closer to the real optimal SJIH than any random guess. In our example query processing environment, we let Query 1 be the most important query. We then repeat the process of finding the next guess with lower query processing costs. This process is looped until no more new guess with lower cost can be found. In each iteration, all SJIHs resulting from the following five possible ways of finding the next guess are considered:

- transfer one class from a SJI to another SJI;
- split an existing SJI into two smaller SJIIs;
- include a class not included in any SJIIs into an existing SJI to form a bigger SJI;
- exclude a class from an existing SJI.

### 6.1.3 Evaluation of the Effectiveness of HCHSA

Let the cardinalities of the classes: **Design**, **Team**, **CompositePart**, **Factory**, **Supplier**, **Engineer**, **Office**, **Documentation**, **AtomicPart**, **Quotation** and **Connection** be 1 Million, 2 Million, 2 Million, 1 Million, 1 Million, 4 Million, 2 Million, 2 Million, 8 Million, 4 Million and 16 Million, respectively. The object sizes of all the classes are set to be 400 bytes\(^2\). Regarding the forward and reverse fan-out values:

- **Design-Team**, **Design-CompositePart**, **Team-Engineer** and **AtomicPart-Connection** links \(f=2/r=1\)
- **Design-Factory** and **Design-Supplier** links \(f=2/r=2\)
- **Team-Office** and **CompositePart-Documentation** links \(f=1/r=1\)
- **CompositePart-AtomicPart** and **Supplier-Quotation** links \(f=4/r=1\)

Regarding the query processing environment, we have 3 sets of queries. The first set consists of the two queries as shown in Figures 99 and 100, each accessing 5 classes. The second set consists of 7 queries, each accessing 2 classes. Finally, the third set consists of all the queries in both set 1 and set 2. The motivation for this setup is to have different sets of queries (accessing a larger number of classes, a smaller number of classes, and a mix of both) to observe the effectiveness of the HCHSA under different query processing environments. The selectivities of the queries are set to be 0.1. Similar results are obtained with other selectivities.

---

1. By most important query, we mean a query which is executed most frequently, the query requesting shortest response time, and/or the query with the highest query processing cost.
2. Similar results are obtained for different object sizes. The larger the object size, the better is the use of SJIH when compared with no indices.
Results

In the first part of the experiment, we use Exhaustive Enumeration Algorithm (EEA) to exhaustively enumerate all possible SJIHs to find the minimum cost SJIH as shown in Figure 104. This absolute minimum cost SJIH serves as a measure to the quality of the output of the HCHSA. Figure 105 shows the scatter plot for storage space required by the SJIH vs. the total query processing cost of all the possible SJIHs involved in the EEA for the first set of queries. Similar results are obtained for the other two sets of queries. As the leftmost branch is the most complicated and interesting one, we concentrate on this branch. The result plots for other branches are similar.

In the second part of the experiment, we use our HCHSA with the initial SJIH as shown in Figure 103 (which just covers all the classes in Query 1). The HCHSA produces the same final SJIH as the EEA, which means the HCHSA finds the optimal solution in its hill-climbing process. In Figure 106, we also present the scatter plot for storage space vs. the total cost of all the possible SJIHs involved in the HCHSA.

Observations/Discussion

We now discuss the results in terms of effectiveness and efficiency obtained in Figures 105 and 106.

Effectiveness

- HCHSA finds the optimal solution in all three sets of queries, which means HCHSA is effective when compared with the EEA.
- From the following table of query processing cost and storage space values, we conclude that the use of the optimal SJIH is highly effective when compared with the case with no indices (which rely on pointer traversal).

<table>
<thead>
<tr>
<th></th>
<th>Total query processing cost for query Set 1 (in unit of 8KB disk page accesses)</th>
<th>Storage space required for the indices in query Set 1 (in unit of 8KB disk pages)</th>
<th>Total query processing cost for query Set 2</th>
<th>Storage space in query Set 2</th>
<th>Total query processing cost for query Set 3</th>
<th>Storage space in query Set 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>No indices</td>
<td>1,404,811</td>
<td>0</td>
<td>1,128,531</td>
<td>0</td>
<td>2,533,342</td>
<td>0</td>
</tr>
<tr>
<td>Initial SJIH (HCHSA)</td>
<td>81,116</td>
<td>53,014</td>
<td>688,232</td>
<td>6,994</td>
<td>27,902</td>
<td>53,014</td>
</tr>
<tr>
<td>Final SJIH (HCHSA)</td>
<td>48,949</td>
<td>48,829</td>
<td></td>
<td></td>
<td>53,015</td>
<td>60,141</td>
</tr>
<tr>
<td>Optimal SJIH (EEA)</td>
<td>48,949</td>
<td>48,829</td>
<td>6,994</td>
<td>53,015</td>
<td>53,015</td>
<td>60,141</td>
</tr>
</tbody>
</table>
Efficiency

- The efficiency of the algorithm to find the optimal SJIH is dictated by the number of SJIHs probed during the search. As illustrated in the following table, the HCHSA is highly efficient when compared with the EEA.

<table>
<thead>
<tr>
<th></th>
<th>Number of SJIHs probed during the search for query Set 1</th>
<th>Number of SJIHs probed during the search for query Set 2</th>
<th>Number of SJIHs probed during the search for query Set 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>HCHSA</td>
<td>91</td>
<td>195</td>
<td>115</td>
</tr>
<tr>
<td>EEA</td>
<td>17,901</td>
<td>17,901</td>
<td>17,901</td>
</tr>
</tbody>
</table>

- We further note that the process of branch separation greatly reduces the number of SJIHs to be probed in the later stages of the algorithm. For a query graph (without branch separation) with 11 classes, the number of SJIHs probed in the EEA will be approximately $3 \times 10^{76}$ as compared with 17,901 in our case\(^1\). Note that after branch separation, the query graph has only 5 classes.

6.1.4 Index selection with limited storage space

In the case where there is only limited storage space available for storing the indices, from the lower envelop of the scatter plot of storage space vs. total cost as shown in Figure 106, HCHSA facilitates selection of a near-optimal set of indices for efficiently executing the set of queries. Note that each point in the scatter plot corresponds to a set of indices that execute the given set of queries with a specific storage cost and a specific query processing cost. For a fixed storage space value (fixing x-axis value), if it is less than the storage space required by the optimal SJIH, we can consult the scatter plot to obtain the minimum cost SJIH within the allowed storage space. Otherwise, we use the optimal SJIH.

\(^1\) Refer to Section 5.2.6 on the calculation of number of possible SJIHs in an OODB schema.
6.2 Experiment with Random Queries

In this section, we describe the results of applying HCHSA towards query processing environment with random queries. We compare the efficiency and effectiveness between the two algorithms: Exhaustive Enumeration Algorithm (EEA) and HCHSA.

6.2.1 Experimental Setup

The setup for this experiment is similar to the previous experiment in Section 6.1, except:

- The number of classes accessed in a query is randomly selected from the ranges: 2, 4, 6 and 8.
- Due to the huge search space for EEA, we compare results from three different cases:
  - case (1) for SJIHs with only 1 SJI;
• **case (2)** for SJIHs with 2 SJIJs; and

• **case (3)** for SJIHs with 3 SJIJs.

• We use the forward/backward fan-out values as in the CAD example schema. Furthermore, these is a scale factor \( sf \) to scale up/down the fan-out values. In this section, we report on results using scale factor value of 2.0. The results for other scale factor values follow the same trend.

• In each run, we randomly generate the queries (the results presented below is for number of queries equals to 8 (fixed)\(^1\), the number of classes retrieved in a query is randomly set to either 2, 4, 6 or 8), and the classes are randomly selected. For the above three cases: we first execute the EEA to find the optimal EEA SJIHs, after that, we start with the initial SJIH and then we perform HCHSA to find the final optimal or near-optimal SJIH.

### 6.2.2 Results

The results from a 50-runs experiment (with a total of 400 queries) are as follows:

**Report on the Efficiency of HCHSA vs. EEA**

<table>
<thead>
<tr>
<th>Number of SJIJs in the SJIH</th>
<th>Average number of iterations required by HCHSA</th>
<th>Number of iterations required by EEA</th>
<th>Ratio (HCHSA/EEA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
<td>120</td>
<td>0.1917</td>
</tr>
<tr>
<td>2</td>
<td>129</td>
<td>7,260</td>
<td>0.0178</td>
</tr>
<tr>
<td>3</td>
<td>259</td>
<td>288,100</td>
<td>0.0009</td>
</tr>
</tbody>
</table>

From the above table, for case (1) with one SJI, the hill climbing algorithm uses only 19% of iterations of that of EEA, to find the optimal or near-optimal SJIH. And HCHSA uses far fewer iterations than that of EEA for case (2) and (3) (with 1.78% and 0.09%, respectively). The conclusion is that HCHSA is superior in terms of efficiency in finding the optimal or near-optimal SJIH.

**Report on the Cost Ratio between the HCHSA/EEA at different number of SJIJs**

<table>
<thead>
<tr>
<th>Number of SJIJs in the SJIH</th>
<th>Average Cost Ratio (HCHSA/EEA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0107</td>
</tr>
<tr>
<td>2</td>
<td>1.0668</td>
</tr>
<tr>
<td>3</td>
<td>1.2512</td>
</tr>
</tbody>
</table>

From the second column (Average Cost Ratio (Hill/EEA)) of the above table, we note that on av-

---

1. We have performed experiments with other number of queries, the results for them all follow the same trend.
erage:

- For case (1), HCHSA finds optimal or near-optimal SJIH that costs 1.0107-1.0=1% more expensive than that of EEA. Implying for this case, in terms of effectiveness, HCHSA is comparable with that of EEA.
- For case (2), HCHSA finds SJIH that costs 1.0668-1.0=7% more expensive than that of EEA. Implying HCHSA is still comparable to that of EEA.
- For case (3), HCHSA finds SJIH that costs 1.2512-1.0=25% more expensive than that of EEA. Implying HCHSA is still close to that of EEA, in terms of effectiveness.

We tabulate the cost ratios between the HCHSA and EEA for the three cases below:

<table>
<thead>
<tr>
<th>Number of SJIs in the SJIH</th>
<th>CR=1.0</th>
<th>1.0&lt;CR&lt;=1.5</th>
<th>1.5&lt;CR&lt;=2.0</th>
<th>2.0&lt;CR&lt;=2.5</th>
<th>2.5&lt;CR&lt;=3.0</th>
<th>3.0&lt;CR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>88%</td>
<td>12%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>2</td>
<td>78%</td>
<td>18%</td>
<td>2%</td>
<td>2%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>3</td>
<td>48%</td>
<td>28%</td>
<td>22%</td>
<td>2%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

From the above table, we note that:

- For case (1), 88% of the times HCHSA finds an optimal SJIH with the same cost as that of EEA. Further we note that HCHSA only uses 19% number of iterations/steps when compared with EEA, we therefore concluded that for case (1), HCHSA is superior to EEA both in terms of effectiveness and efficiency.
- For case (2), 78% of the times HCHSA finds the optimal SJIH. 18% of the times HCHSA finds a final SJIH with cost between 1.0 to 1.5 of that of EEA. And the other 4% of times with cost between 1.5 to 2.5 of that of EEA. We note that HCHSA uses only 1.78% number of iterations/steps as compared with EEA. In this case, we conclude that in terms of effectiveness, HCHSA is close to that of EEA. But in terms of efficiency, HCHSA is superior to that of EEA.
- For case (3), 48%, 28%, and 22% of the times HCHSA finds a final SJIH with cost equals to that of EEA, between 1.0-1.5, and 1.5-2.0, respectively. We note that HCHSA uses only 0.09% number of iterations as compared with EEA. In this case, we conclude that in terms of effectiveness, HCHSA is close to that of EEA. But in terms of efficiency, judging from the relatively very low number of iterations required, HCHSA is again superior to that of EEA.

6.3 Experiment on HCHSA Performance Gain

In this experiment, we concentrate on the performance gain of HCHSA. We vary the maximum number of SJIs in the SJIH and observe the performance gain between the initial and final SJIHs.
6.3.1 Experimental Setup
The experimental setup is identical with the previous experiment in Section 6.2.

6.3.2 Results
We vary the maximum number of SJIs in the SJIH and we report on the performance gain between the initial and final SJIHs (i.e., the average ratio of final cost/initial cost). The averages are over 50 run with 8 queries each (a total of 400 queries).

Due to the large number of possible SJIHs, we experiment with queries that access the top 7 classes in the left-most branch of the extended CAD schema. With an OODB schema with 7 classes, the maximum number of SJIs in a SJIH without redundancy is 6. An example of such SJIH (with 6 SJIs) is the BSJIH—with all 6 BJIs between every two neighbouring classes. By redundancy in a SJIH, we mean that no SJI (in the SJIH) is completely contained in another SJI, which waste storage spaces and has no improvement in performance.

Report on the performance gain between the initial and the final SJIH of HCHSA

<table>
<thead>
<tr>
<th>Maximum Number of SJIs in the SJIH</th>
<th>Average Final SJIH cost of HCHSA</th>
<th>Average Initial SJIH cost of HCHSA</th>
<th>Average Ratio (Final Cost/Initial Cost)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,958,448</td>
<td>2,303,234</td>
<td>0.8576</td>
</tr>
<tr>
<td>2</td>
<td>1,034,429</td>
<td>2,303,234</td>
<td>0.4557</td>
</tr>
<tr>
<td>3</td>
<td>862,271</td>
<td>2,303,234</td>
<td>0.3840</td>
</tr>
<tr>
<td>4</td>
<td>841,761</td>
<td>2,303,234</td>
<td>0.3752</td>
</tr>
<tr>
<td>5</td>
<td>819,276</td>
<td>2,303,234</td>
<td>0.3655</td>
</tr>
<tr>
<td>6</td>
<td>768,037</td>
<td>2,303,234</td>
<td>0.3444</td>
</tr>
</tbody>
</table>

![Plot of Max no of SJIs vs. Final Cost/Initial Cost](image)

Figure 107: Plot of Maximum number of SJIs vs. Final Cost/Initial Cost

From the above table and plot, which are averaged over 50 runs, the observations are:

- As the maximum allowable number of SJIs in the SJIH increases, the ratio of final cost/initial
cost decreases. The ratio decreases rapidly first then flatten as maximum number of SJIIs reaches 3 or above.

- The performance gain varies from 1-0.86=14% (with one SJI in the SJIH) to 1-0.34=66% (with maximum 6 SJIIs in the SJIH). The conclusion is HCHSA obtained good gains between the initial SJIH towards the final SJIH.

**Report on the efficiency of HCHSA**

<table>
<thead>
<tr>
<th>Maximum Number of SJIIs in the SJIH</th>
<th>Average Number of Steps/Iterations required by HCHSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td>128</td>
</tr>
<tr>
<td>3</td>
<td>258</td>
</tr>
<tr>
<td>4</td>
<td>317</td>
</tr>
<tr>
<td>5</td>
<td>373</td>
</tr>
<tr>
<td>6</td>
<td>386</td>
</tr>
</tbody>
</table>

![Plot of Max no of SJIIs vs. No of Steps](image)

Figure 108: Plot of Maximum number of SJIIs vs. No of Steps

From the above table and plot, which are averaged over 50 runs, the observations are:

- As the maximum allowable number of SJIIs in the SJIH increases, the number of steps/iterations required by the HCHSA increases. The number of steps increases rapidly first then flatten as maximum number of SJIIs reaches 5 or above.

- The number of steps varies from 24 (with one SJI) to 386 (with maximum 6 SJIIs in the SJIH). That means HCHSA requires at most 386 steps, but EEA with only three SJIIs requires 288,100 steps. The conclusion is that HCHSA is superior in terms of efficiency as compared with EEA.
6.4 Integration of VCP Techniques into SJIH

Within the SJIH framework, the Complete-SJIH (CSJIH) is the most powerful since all the OID
information and hence all the relationships between objects in all the classes is captured. However,
from the results of Chapter 5’s analytical experiments, the CSJIH will have high storage cost at a
high degree of sharing between the classes of the OODB schema and hence high index retrieval
cost, making CSJIH not suitable for complex object retrieval. In this case, a simpler PSJIH will be
employed. In the following we use the extended CAD schema for further illustration. The CSJIH
will be a collection of tuples with the OIDs from all the classes in the whole schema:

\[ \langle D[], T[], E[], O[], C[], M[], A[], N[], F[], S[], Q[] \rangle \]

where "\( \langle \rangle \)" is the tuple constructor, and \( D[], T[], E[], O[], C[], M[], A[], N[], F[], S[] \) and \( Q[] \) are
the OIDs of classes: Design, Team, Engineer, Office, CompositePart, Documentation, AtomicPart,
Connection, Factory, Supplier and Quotation, respectively. The tuple in the CSJIH resembles an
unpartitioned class with the above OIDs as instance variables. At a high degree of sharing, the total
number of tuples will be high. We need to use a simpler PSJIH and to split/vertically partition the
original tuples of the CSJIH into simpler tuples with fewer OIDs. There are three cases with dif-
ferent guidelines for the vertical partitioning: (1) unconstrained pair-up branches, (2) high degree
of sharing between classes, and (3) query access patterns.

**Unconstrained pair-up branches**

From section 5.2.3, we note that the number of tuples in a SJI increases rapidly if classes from un-
constrained pair-up branches are included in the SJI. For example, if the branches: Design-Factory,
and Design-Supplier-Quotation are unconstrained branches, then we can form a SJIH with the fol-
lowing three SJIIs:

- **SJI1**—an SJI with all the classes from the left-most branch of the extended CAD schema, that
  means \( SJI1 \) has the following tuple structure:

  \( SJI1 = \langle D[], T[], E[], O[], C[], M[], A[], N[] \rangle \)

- **SJI2**—an SJI with the two classes in the Design-Factory branch

  \( SJI2 = \langle D[], F[] \rangle \)

- **SJI3**—an SJI with the three classes in the Design-Supplier-Quotation branch

  \( SJI3 = \langle D[], S[], Q[] \rangle \)

After vertical partitioning of the original CSJIH, we have a new PSJIH with three SJIIs, but
now all the unconstrained pair-up branches are removed. From the storage and index retrieval cost
model for SJIH (developed in Chapter 5), there will be lower storage and query processing cost.
Finally, we observe that the three SJIIs are similar to the vertical fragments after vertical partition-
ing the original CSJIH tuple. The new PSJIH will have much lower storage cost and index retrieval
cost, and hence will better support more efficient complex object retrieval.
High degree of sharing

From section 5.2.3, we also note that the number of SJIH tuples increases rapidly if the degree of sharing between the classes is high. For example, if the degree of sharing/forward fan-out between the classes CompositePart and AtomicPart is high, there will be a rapid increase in the number of SJI tuples if both classes are included in the same SJI, an example is SJI1 (as mentioned in (1)). In this case, we can isolate the two classes to form a new SJI, and after vertically partitioning SJI1, it will be partitioned into three smaller SJJIs (namely SJI1-1, SJI1-2 and SJI1-3, respectively) with

\[ SJI1-1 = \langle D[], T[], E[], O[], C[], M[] \rangle \]
\[ SJI1-2 = \langle C[], A[] \rangle \]
\[ SJI1-3 = \langle A[], N[] \rangle \]

After the vertical partitioning, the two classes with a high degree of sharing are isolated from the original SJI to form a new BJ. This prevents the multiplying effect of the high fan-out from affecting the number of SJI tuples in the other two SJJIs--SJI1-1 and SJI-3. From the storage cost and index retrieval cost model, the resulting SJIH (with the three SJJIs--SJI1-1, SJI1-2 and SJI1-3) will have much lower storage and index retrieval cost, even at a high degree of sharing, and hence will better support more efficient complex object retrieval.

Query access patterns

The third way of vertically partitioning an SJIH is guided by the query access patterns. By means of vertical partitioning guided by query access pattern, those OIDs that are frequently accessed together are grouped into the same SJI/vertical fragment. This has the same effect as our VCP technique as discussed in Chapter 3.

As both VCP and SJIH are based on the notion of complex object, there are a number of similarity between VCP and SJIH: we observe that the original CSJIH is like our unpartitioned class in the VCP, and the SJJIs in the PCSIH are like our vertical fragments. Further, the splitting the CSJIH/complex SJIH into a number of smaller SJJIs is like vertical class partitioning. Hence the VCP and SJIH techniques are complementary in that VCP technique can be applied to SJIH or vice versa. Let us illustrate how VCP technique can be apply to SJIH by the following analytical experiment.

Experimental Setup

The database schema, cardinalities, and forward/backward fan-outs are the same as in section 6.1's extended CAD schema. We concentrate on the left-most branch as shown in Figure 109. The other two branches (middle and right-most) are unconstrained pair-up branches, and hence the middle and right-most branches will be separated (as suggested by (1) above). The selectivity is 0.1 and the scale factor ranges from 0.125 to 8.0. The queries used in this experiment are illustrated in Table 17.
Table 17: Query Access Patterns

<table>
<thead>
<tr>
<th>Query</th>
<th>D[]</th>
<th>T[]</th>
<th>E[]</th>
<th>O[]</th>
<th>C[]</th>
<th>M[]</th>
<th>A[]</th>
<th>N[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

After the application of the VCP technique, the resulting vertical partitioning scheme/partial SJIIH scheme is: { (D[], T[], C[], O[], M[]) (C[], A[], N[]) (T[], E[]) } and is shown in Figure 110. This means that the PSJIH will contain three SJIIs, the first SJI will contain the OIDs of classes Design, Team, CompositePart, Office and Documentation, and so on.

Figure 109: Complete-SJIH

Figure 110: Vertically Partitioned Complete-SJIH
Results
Table 18 shows the results from the experiment for scale factor equal to 2.0. The results for the other scale factors follow similar trends. Table 19, shows the results for the average Normalized IO (Index retrieval cost of VP case divided by UP case) and Normalized storage cost (Storage cost of VP case divided by UP case).

Table 18: Results for Index retrieval cost

<table>
<thead>
<tr>
<th>Query</th>
<th>Index retrieval cost (Vertically Partitioned case)</th>
<th>Index retrieval cost using CSJIH (Unpartitioned case)</th>
<th>Normalized IO (VP/UP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>22377</td>
<td>1521570</td>
<td>0.0147</td>
</tr>
<tr>
<td>1</td>
<td>313266</td>
<td>1521570</td>
<td>0.2059</td>
</tr>
<tr>
<td>2</td>
<td>61535</td>
<td>1521570</td>
<td>0.0404</td>
</tr>
<tr>
<td>3</td>
<td>1521570</td>
<td>1521570</td>
<td>1.0000</td>
</tr>
<tr>
<td>4</td>
<td>61535</td>
<td>1521570</td>
<td>0.0404</td>
</tr>
<tr>
<td>5</td>
<td>61535</td>
<td>1521570</td>
<td>0.0404</td>
</tr>
<tr>
<td>6</td>
<td>61535</td>
<td>1521570</td>
<td>0.0404</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td></td>
<td>0.1975</td>
</tr>
</tbody>
</table>

Table 19: Results for Normalized IO and Storage

<table>
<thead>
<tr>
<th>Scale factor</th>
<th>Average Normalized IO (VP/UP)</th>
<th>Normalized Storage (VP/UP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.125</td>
<td>1.5207</td>
<td>7.7008</td>
</tr>
<tr>
<td>0.25</td>
<td>0.6976</td>
<td>2.9120</td>
</tr>
<tr>
<td>0.5</td>
<td>0.3866</td>
<td>1.2206</td>
</tr>
<tr>
<td>1.0</td>
<td>0.2563</td>
<td>0.5515</td>
</tr>
<tr>
<td>2.0</td>
<td>0.1975</td>
<td>0.2610</td>
</tr>
<tr>
<td>4.0</td>
<td>0.1696</td>
<td>0.1268</td>
</tr>
<tr>
<td>8.0</td>
<td>0.1561</td>
<td>0.0625</td>
</tr>
</tbody>
</table>

Discussion
From Table 17, we observe that OIDs of classes Design, Team, CompositePart, Office and Documentation are frequently accessed together. After vertical partitioning of the CSJIH, we obtain an PSJIH with three SJIIs. We group these OIDs into the same fragment. From Table 18, we observe that the average Normalized IO is 0.1975, implying a cost saving of 1-0.2=80% after performing the vertical partitioning on the CSJIH, which is a substantial cost saving. Table 19 shows the trends
of both average Normalized IO and Normalized Storage vs. the variation of scale factor. We observe that as the scale factor increases, the forward/backward fan-out values also increase, causing a high increase in both the storage and index retrieval cost. These lead to the decreasing trends of the average Normalized IO and Normalized storage. As shown in the Average Normalized IO column of Table 19, except for very small scale factor (0.125), all the averages are smaller than 1.0, implying the application of VCP techniques to SJIH decreases the index retrieval cost. Furthermore, from the Normalized Storage column, except for low scale factor (0.125 to 0.5), all the PSJIHs have lower storage cost after applying VCP to the CSJIH. In the case of low scale factor, from the storage and index retrieval cost model, CSJIH is already efficient, without the help of the VCP technique. Finally, we conclude that the VCP technique can help in improving the performance of SJIH in supporting complex object retrieval.

6.5 Summary

In this chapter, a heuristic algorithm HCHSA is developed to select appropriate indices to efficiently process a given set of queries. Our results show that, given a set of queries and limited index storage space, the heuristic algorithm facilitates fast selection of a near-optimal set of indices for efficiently executing the queries. In conclusion, our HCHSA is highly efficient when compared with the EFA. We further note that there is a high performance gain between the initial and final SJIH of the HCHSA. Finally, we demonstrated that the VCP techniques can be integrated into the SJIH framework and facilitate efficient complex object retrieval.

We note that for an OODB schema with a large number of classes (when compared with the analytical experiment settings of this thesis), the search space is much larger and there is a higher chance that the hill-climbing algorithms may not find the optimal solution, but may be trapped in a local minima.
Chapter 7

Conclusions and Future Work

7.1 Summary of Research Contributions

7.1.1 Vertical Class Partitioning

Vertical class partitioning in object oriented databases (OODBs) is a challenging and relevant problem. Though similar problems have been addressed in relational database systems, the complexity of OODB models involving class inheritance hierarchy and class composition hierarchy complicates the problem and thus requires a new approach to be developed. Vertical class partitioning in OODBs improves the efficiency of executing a given set of queries by grouping the instance variables of a class into non-overlapping subsets, with the aim of reducing irrelevant data (instance variable) accesses. The query processing efficiency is enhanced when a query can access fewer instance variables (projection ratio). The effect of indexing depends on the number of object instances retrieved for a query (selectivity factor). In order to study the effect of vertical class partitioning, we have developed a comprehensive analytical cost model for calculating the number of disk accesses required for processing a query. We consider an OODB schema with both class inheritance hierarchy and class composition hierarchy, both sequential and index access schemes, and over a range of selectivity values and projection ratios. This represents the first piece of work on an the analytical cost-based approach advocated for vertical class partitioning in OODBs.

Our analytical cost model has been used to study the effectiveness of vertical partitioning under various experimental setups. First, we showed that for a given query work load, there is an optimal vertical partitioning scheme. Second, we demonstrated that the higher the fan-out the better the savings from vertical fragmentation. Third, we showed that the percentage of savings due to vertical fragmentation is constant as the cardinality of the class increases. Fourth, we verified that
vertical class partitioning is better for small projection ratios. We also studied the trade-off between the selectivity factor and projection ratio for evaluating the goodness of vertical partitioning versus clustered index on a root class. We have found that for a large parameter space vertical class partitioning out performs a clustered index for high selectivity and not so high projection ratio. Thus there are cases in which it is better to apply vertical class partitioning than a clustered index.

Vertical class partitioning is more suitable to be applied to a production type OODB system with rather static database and query processing characteristics. For an evolving type of OODB system with highly dynamic database and query processing characteristics, vertical class partitioning can still be applied, but we need extra processing to monitor for the need of reorganizing the existing vertical class partitioning scheme.

Due to the encapsulation property of OODBs, methods govern the nature of optimal (or near optimal) vertical partitioning schemes. In this thesis, we have developed, based on MDGs (method dependency graphs), a general purpose cost model for method execution, and applied this model to come up with vertical partitioning algorithms. As data in an OODB is accessed through methods, With the help of method transformation, our method-induced vertical class partitioning technique can easily support fragmentation transparency. This fragmentation transparency support facilitate the integration of method-induced vertical class partitioning technique into an OODBMS.

Two algorithms have been developed to exploit method execution for deriving optimal or near-optimal vertical partitioning schemes. The first algorithm is a cost-driven algorithm which uses a cost model for method execution to exhaustively search all the partitioning schemes so as to find the optimal vertical partitioning scheme. This algorithm is useful for comparing the effectiveness of the other algorithms, but may not be practical since it has a high cost of $O(n^n)$ where $n$ is the number of instance variables. The second algorithm, HCHA, first applies an affinity-based algorithm [64] to get an initial partitioning scheme, and then applies a hill-climbing heuristic to improve this solution by using the cost model for method execution. It is shown that the HCHA (which is significantly more efficient than the cost-based approach) generates mostly optimal or near optimal VCP schemes. The HCHA approach thus represents a good compromise, generating a solution that is at least as good as the solution provided by the affinity-based approach, and in most cases, can yield the optimal partitioning scheme. Thus in this thesis, we took into consideration the methods being invoked during OODB processing to generate optimal or near-optimal vertical class partitioning schemes. But as with all hill-climbing algorithms, there are chances that the algorithm cannot generate the optimal solution and be trapped in a local minima.

7.1.2 Structural Join Index Hierarchy
Queries in OODB systems typically retrieve complex objects or their component objects. In order to support their efficient execution it is important to provide suitable access methods for complex
object retrieval. Earlier work on OODB indexing has addressed efficient navigation through path expressions in OODBs. In this thesis, we have extended existing work by advocating a structural join index hierarchy for facilitating complex object retrieval. A structural join index hierarchy (SJIH) is a sequence of OIDs which provides direct access to component objects of a complex object. Three types of SJIHs are studied: the Complete-SJIH provides direct access to all the OIDs of the component objects of a complex object; the Base-SJIH requires a join of all the base join indices to retrieve the complete complex object; and the Partial-SJIH provides direct access to the most frequently accessed component objects. Our performance results demonstrated the utility of the three SJIHs, and showed the superiority of selecting SJIH over other indexing schemes, such as Nested Index and Multi-index. Finally, a heuristic algorithm is developed to select appropriate indices to efficiently process a given set of queries. Our results show that SJIH indexing mechanisms not only facilitate efficient retrieval of complex objects, but also outperform many of the existing indexing mechanisms for complex object retrieval. Moreover, given a set of queries and a limited index storage space, the heuristic algorithm facilitates fast selection of a near-optimal set of indices for efficiently executing the queries. We further show how vertical class partitioning techniques can be employed to further reduce storage and retrieval costs for SJIH.

7.2 Implementation of VCP and SJIH in Commercial OODBMS

An interesting issue is the incorporation of our performance oriented techniques into commercial Object Oriented Database Management Systems (OODBMS). Our viewpoint is that, as vertical fragments are represented as classes, this uniform approach facilitates the implementation of vertical class partitioning support into commercial OODBMS. Furthermore, in an OODB, data is accessed through methods. Our approach of method-induced VCP will facilitate the OODBMS in supporting fragmentation transparency.

Our SJIH subsumes previous OODB indexing methods and provides flexible support for complex object retrieval. The general nature of SJIH simplifies its implementation in OODBMSs. Previously, we needed to implement different sets of system routines to support different OODB indexing methods. With SJIH, we need to implement a single set of system routines, that facilitate different indexing schemes for supporting complex object retrieval for different query processing requirements.

7.3 Future Work

Future work includes the following topics:

- **Index selection algorithm in dynamic query processing environments:**
  
  In a dynamic query processing environment the class access patterns change over time. There-
fore static (fixed) SJIH cannot execute queries efficiently over time. In order to deal with
dynamic changes, the OODB indexing framework should be extensible and flexible. Our repre-
sentation of SJIH provides facilities for (i) building complex indices by combining primitive
indices, and (ii) generating primitive indices by projecting over complex indices. An important
research issue to address is to develop an adaptive SJIH selection methodology to cater for
dynamic query processing environments.

- **Applying VCP and SJIH to Web system design:**
  Web pages contain different elements, (like, text, graphics, audio, and video). Usually these
elements are not accessed together. Some of them are more frequently accessed. This locality
in data accesses implies that the use of VCP for Web page design may be beneficial and may
effectively reduce irrelevant data accesses. Further, SJIH can be applied to retrieve a set of
related (linked) Web pages for a given user request.

- **Applying VCP and SJIH to Data Warehouse design:**
  A Data Warehouse contains a huge amount of data ready for on-line analytical processing
(OLAP). The major difficulty faced by Data Warehouse designers is the large amount of disk
IOs required to obtain the results for on-line analysis. An interesting issue to address is the
applicability of VCP and SJIH to Data Warehouse design so as to improve the throughput of
OLAP.

- **Applying our cost-based vertical class partitioning approach to horizontal class partition-
ing in OODBs:**
  Horizontal class partitioning (HCP) is also a hard problem [68,49]. Similar to VCP, the total
number of possible HCP partitioning schemes is huge. Our viewpoint is that, similar to VCP,
with the help of an analytical cost model, a cost-based HCP algorithm can both be effective and
efficient in finding the cost optimal or near-optimal partitioning scheme.

- **Investigating on the number of cost model parameters:**
  To utilize our VCP and SJIH effectively, we require high quality estimation of cost model
parameters. An interesting issue to address is to reduce the number of cost model parameters,
while maintaining the effectiveness of our VCP and SJIH techniques. This may be done by
making use of heuristics.

- **Investigating on query processing strategy that make use of both the VCP and SJIH tech-
niques:**
  Given a vertically partitioned OODB schema, we may apply SJIH to the vertical class frag-
ments to further enhance complex object retrievals, or vice versa. An interesting issue to
address is to develop query processing strategy that can make use of both the VCP and SJIH
techniques to further expedite query processing.
• Extending SJIH framework to index on class inheritance hierarchies:
  Our SJIH framework subsumes the previous work in OODB indexing for class composition hierarchies. An interesting issue to address is to extend SJIH to cater for query processing that involves the class inheritance hierarchies.
References


[26] A. A. Diwan, Sanjeeva Rane, S. Seshadri and S. Sudarshan, "Clustering Techniques for Min-
imi dong External Path Length", in Proc. Intl. Conf. on Very Large Data Bases, pages 342-
353, 1996.

[27] C. Ezeife and K. Barker, "Horizontal class fragmentation in a distributed object based sys-

[28] C. Ezeife and K. Barker, "Vertical class fragmentation in a distributed object based system", 

Induced Vertical Partitioning Algorithms", Technical Report, HKUST96-33, Department 

Object-Oriented Databases", in Proc. of the Fifth Intl. Conf. on Database Systems for
Advanced Applications (DASFAA'97), Melbourne, Australia, pages 11-20, April 1-4, 
1997. Also as an extended version technical report HKUST-CS96-32, HKUST, available

Query Processing in Object Oriented Databases", submitted to IEEE Transactions on
Knowledge and Data Engineering, 1997.

Efficient Complex Object Retrieval", in Proc. of 5th International Conference on Founda-

[33] C. W. Fung, K. Karlapalem and Q. Li, "Index-based Complex Object Retrieval in Object-Ori-
ented Databases: Mechanisms, Evaluation and Selection", working paper, to be submitted

[34] C. W. Fung, K. Karlapalem and Q. Li, "Cost-Driven Vertical Class Partitioning for Methods
in Object-Oriented Databases", working paper, to be submitted to Very Large Data Bases

[35] C. W. Fung and Q. Li, "Versatile Querying Facilities for a Dynamic Object Clustering
Model", in Proc. of the Fourteenth International Conference on Object-Oriented & Entity
Relationship Modelling (O-O ER'95), Gold Coast, Australia, pages 77-88, Dec 1995.

in Object-Oriented Databases", accepted for publication in the International Journal of


[73] B. Sreenath and S. Seshadri, "The hC-tree: An Efficient Index Structure for Object Oriented Databases", in Proc. 20th Intl. Conf. on Very Large Data Bases, pages 203-213, 1994.


[75] H. K. Wong and K. W. Yiu, "Vertical Partitioning in Object-Oriented Databases", Undergraduate Final Year Project, Department of Computer Science, Hong Kong University of Science & Technology, Dec 1996.

[76] Z. Xie and J. Han, "Join Index Hierarchies for Supporting Efficient Navigations in Object-Oriented Databases", in Proc. Intl. Conf. on Very Large Data Bases, pages 522-533, 1994.


Publications of Chi-wai Fung


C. W. Fung and Q. Li, "Versatile Querying Facilities for a Dynamic Object Clustering Model", in Proc. of the Fourteenth International Conference on Object-Oriented & Entity Relationship Modelling (O-O ER'95), Gold Coast, Australia, pages 77-88, Dec 1995.

Appendix A

[64] Graph-based algorithm

Step 1: Construct the affinity graph of the attributes in the classes
Step 2: Start from any node
Step 3: Select an edge which satisfies the following conditions: it should
      be linearly connected to the tree already constructed and it
      should have the largest value among the possible choices of edges
      at each end of the tree
      /* this iteration will end when all nodes are used for tree
         construction */
Step 4: When the next selected edge forms a primitive cycle:
       If a cycle node does not exist, check for the possibility of a
       cycle and if the possibility exists, mark the cycle as an
       affinity cycle. Consider this cycle as a candidate partition.
       Goto Step 3
       If a cycle node exists already, discard this edge and goto Step 3
Step 5: When the next selected edge does not form a cycle and a candidate
        partition exists:
        If no former edge exists, check for the possibility of extension
        of the cycle by this new edge. If there is no possibility, cut
        this edge and consider the cycle as a partition. Goto Step 3
        If a former edge exists, change the cycle node and check for the
        possibility of extension of the cycle by the former edge. If
        there is no possibility, cut the former edge and consider the
        cycle as a partition. Goto Step 3

[64] graph-based algorithm
Appendix B

Derivations of elementary parameters

For OODB schema with a single path \([77,6]\) the probability of an object in \(C_{j-1}\) which does not make a reference to a particular object in \(C_j\) is:

\[
\frac{\left(\begin{array}{c} |C| - 1 \\ f_{j-1} \end{array}\right)}{\left(\begin{array}{c} |C| \\ f_{j-1} \end{array}\right)}, \text{ where } \left(\begin{array}{c} m \\ n \end{array}\right) \text{ is the number of different combinations of choosing } n \text{ objects out of } m \text{ objects. After simplification, it becomes: } 1 - \frac{f_{j-1}}{|C|}.
\]

The probability of \(m\) objects in \(C_{j-1}\) which do not refer to a particular object in \(C_j\) is: \(\left(1 - \frac{f_{j-1}}{|C|}\right)^m\). The probability of a particular object in \(C_j\) which is referenced by \(m\) objects in \(C_{j-1}\) is: \(1 - \left(1 - \frac{f_{j-1}}{|C|}\right)^m\).

Therefore, the average number of objects in \(C_j\) which are referenced by these \(m\) objects in \(C_{j-1}\) is:

\[|C_j| x \left(1 - \left(1 - \frac{f_{j-1}}{|C|}\right)^m\right).\]

Number of Object References

The average number of distinct objects in \(C_j\) referenced by a set of \(k\) objects in \(C_i\), for any \(i < j\), is:

\[
fwd(i, j, k) = \begin{cases} p(|C|, i+1, f_{i+1}) & \text{if } j = i+1 \\ p(|C|, i+1, fwd(i, j-1, k)) & \text{if } j > i+1 \end{cases}
\]

where \(p(x, y, z) = x \times \left(1 - \left(1 - \frac{y}{z}\right)^x\right)\). Similarly, for any \(i < j\) we have the average number of distinct objects in \(C_i\) referencing a set of \(k\) objects in \(C_j\):

\[
bwd(i, j, k) = \begin{cases} p(|C|, j, r_{j+1}) & \text{if } j = i+1 \\ p(|C|, j, bwd(i, j, k)) & \text{if } j > i+1 \end{cases}
\]
Note also that as \( \frac{f_i}{r_{i+1}} = \frac{\|c_{i+1}\|}{\|c_i\|} \), hence

\[
\text{fwd}(i, i+1, 1) = \|c_{i+1}\| \times \left(1 - \left(1 - \frac{f_i}{\|c_{i+1}\|}\right)^1\right) = \|c_{i+1}\| \times \frac{f_i}{\|c_{i+1}\|} = f_i.
\]

And \( \text{bwd}(i, i+1, 1) = \|c\| \times \left(1 - \left(1 - \frac{r_{i+1}}{\|c\|}\right)^1\right) = r_{i+1}. \)
Appendix C

Dependency of Index retrieval cost on Selectivity

The index tuple of an SJI is smaller than the page size and when we calculate the index retrieval cost, we apply Case 1 of the Yao function. This is a property of the Yao function $Y(k, m, n)$ that, if $k > 2 \times m$, then $Y(k, m, n) = m$ [66]. This implies that if $Sel = \frac{k}{n} \geq \frac{2 \times m}{n}$ (i.e., the selectivity is higher than a threshold), then the index retrieval cost which is a sum of Yao functions, will only be dependent on $m$ and hence the index retrieval cost is independent of the selectivity.